



US 20030154468A1

(19) **United States**(12) **Patent Application Publication**

Gordon et al.

(10) Pub. No.: **US 2003/0154468 A1**(43) Pub. Date: **Aug. 14, 2003**(54) **VERIFIER TO CHECK INTERMEDIATE LANGUAGE****Related U.S. Application Data**(75) Inventors: **Andrew Gordon**, Cambridge (GB);
Donald Syme, Cambridge (GB);
Jonathon Forbes, Bellevue, WA (US);
Vance P. Morrison, Kirkland, WA (US)(63) Continuation of application No. 09/388,176, filed on
~~SEP. 1, 1999~~, now Pat. No. 6,560,774.**Publication Classification**(51) Int. Cl.⁷ **G06F 9/45; G06F 9/44**(52) U.S. Cl. **717/143; 717/146; 717/126**

Correspondence Address:

LEYDIG VOIT & MAYER, LTD
TWO PRUDENTIAL PLAZA, SUITE 4900
180 NORTH STETSON AVENUE
CHICAGO, IL 60601-6780 (US)(57) **ABSTRACT**

Verification of intermediate language code. In one embodiment, a computer-implemented method first verifies meta-data of an intermediate language code for consistency and accuracy, and then verifying the intermediate language code for consistency and accuracy. This latter part in one embodiment is accomplished by performing first a syntactic check of the intermediate language code, and then a semantic check of the intermediate language code.

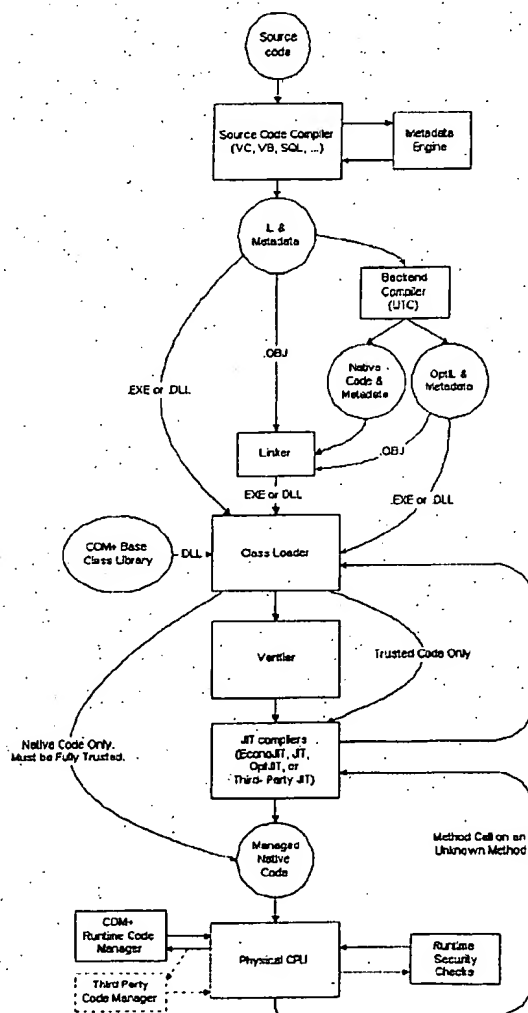
(73) Assignee: **Microsoft Corporation**, Redmond, WA
(US)(21) Appl. No.: **10/370,359**(22) Filed: **Feb. 19, 2003**

FIG. 1

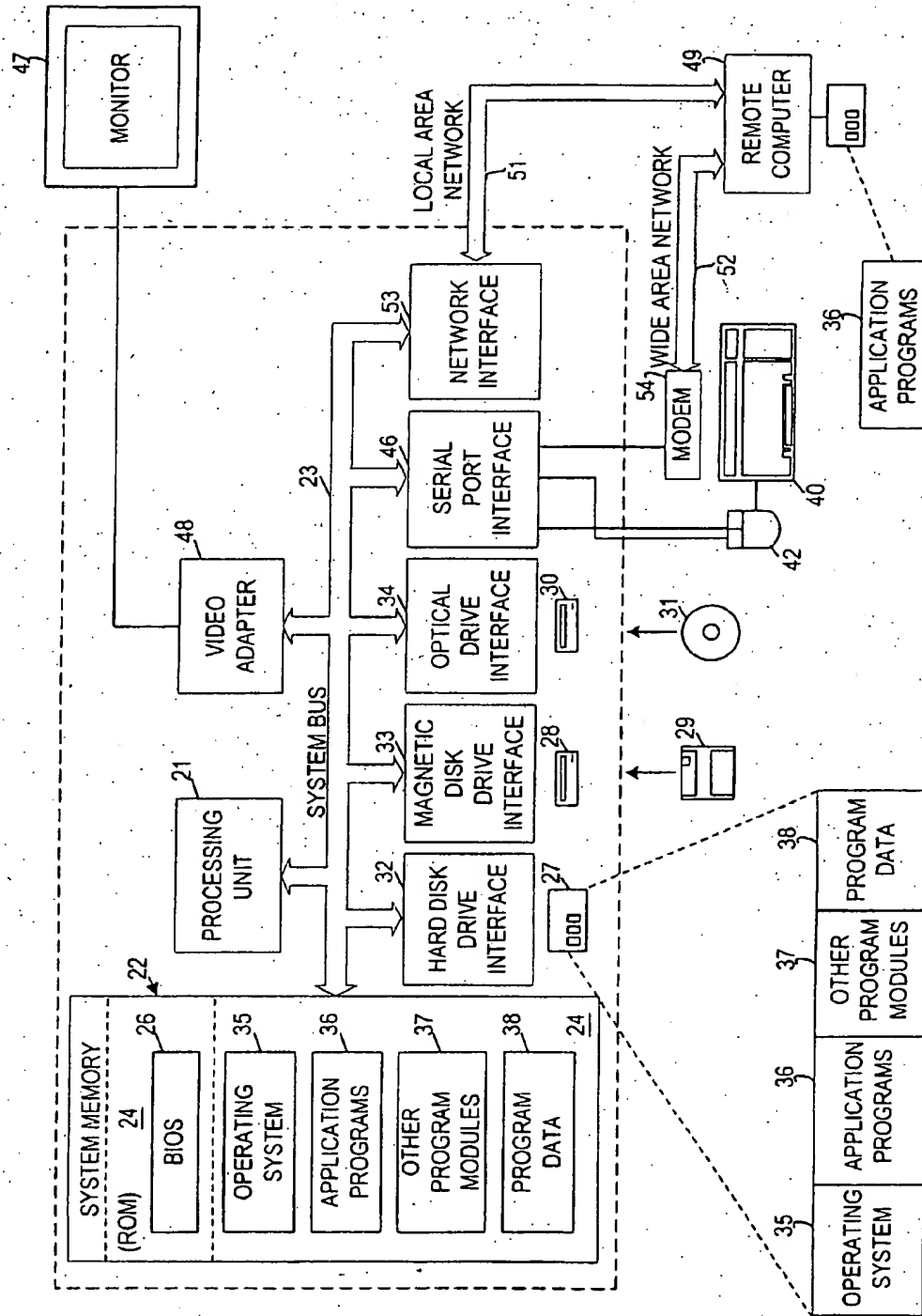


FIG. 2

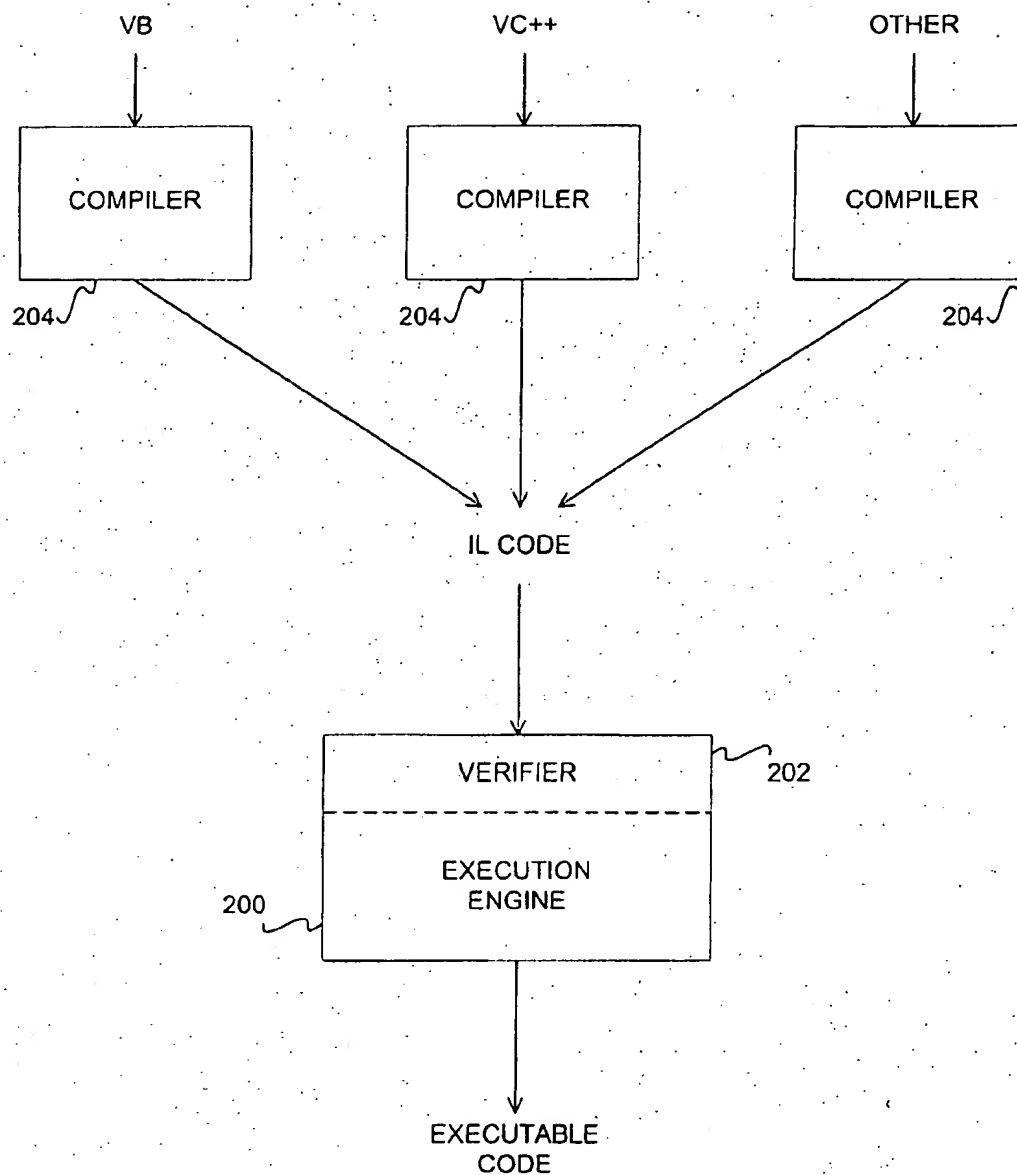
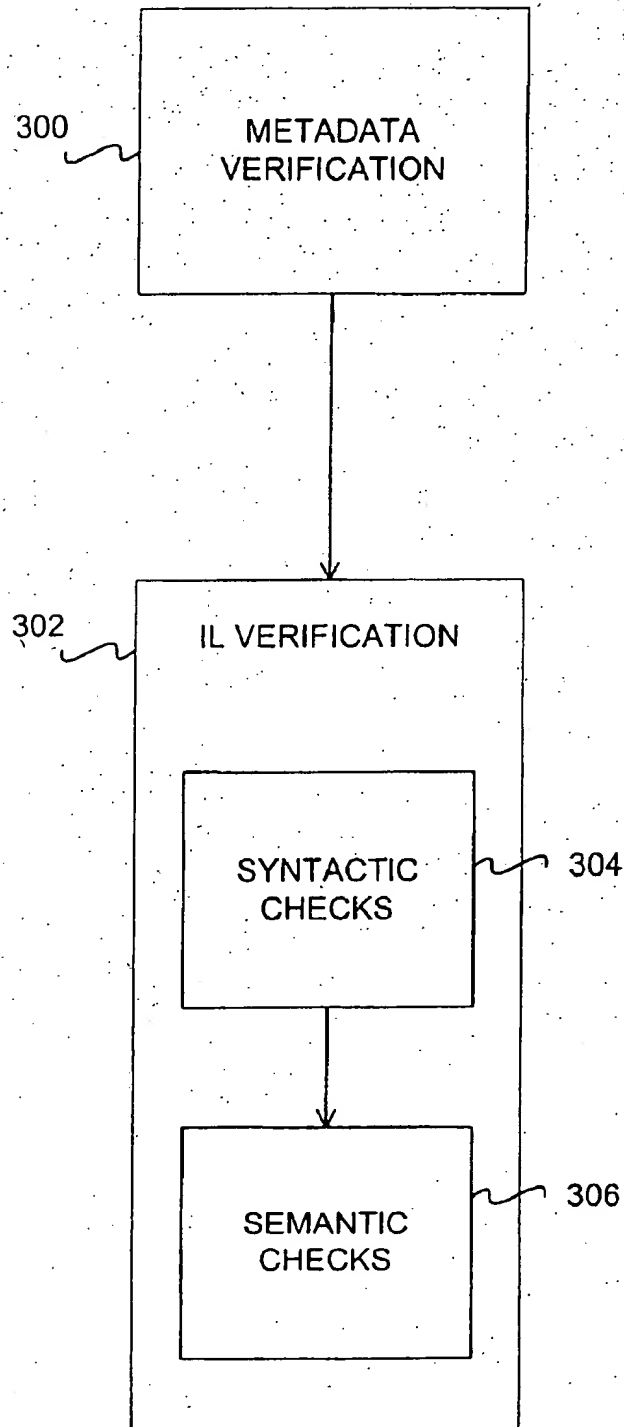


FIG. 3



Flag	Meaning
VER_FLAG_UNINIT	This item is uninitialised. If a byref, it means &(uninitialised <value class or local variable>). If not a byref, it means it is an unconstructed object reference – something that can exist only when verifying a constructor method. It is illegal to have an &(uninitialised objref).
VER_FLAG_BYREF	The item is a byref
VER_FLAG_NULL_OBJREF	The object is the null object reference (thus m_pEEClass does not contain valid data). The component type must be VER_ELEMENT_TYPE_OBJREF.
VER_FLAG_ARRAY	The object is an array of some kind (m_pEEClass provides the general array type, and m_pUnderlyingArrayClass provides the specific type of the array component). The component type must be VER_ELEMENT_TYPE_OBJREF.
VER_FLAG_BYREF_LOCAL	This is the address of a local variable. The local variable number is stored in the upper 16 bits of m_dwFlags. VER_FLAG_BYREF must also be set, and this flag is mutually exclusive with VER_FLAG_BYREF_INSTANCE_FIELD.
VER_FLAG_THIS_PTR	This is used only when verifying constructor methods, where the "this" pointer can be uninitialised. This is so that when the "this" pointer is initialised by calling a constructor, all places where "this" is located (e.g. local variables and other stack locations) can simultaneously be initialised.
VER_FLAG_BYREF_INSTANCE_FIELD	This is the address of an instance field of the class of current method. The instance field number is stored in the upper 16 bits of m_dwFlags. This is used only when verifying value class constructors. VER_FLAG_BYREF must also be set, and this flag is mutually exclusive with VER_FLAG_BYREF_LOCAL.
VER_FLAG_BYREF_PERMANENT_HOME	This is a byref which has a permanent home. This means that it points to something which is acceptable to return to the callee; i.e. something in the gc heap, NOT a local variable or a location on stack. VER_FLAG_BYREF must also be set.

FIG. 4

	Type	Flags	EEClass	UnderlyingArrayEEClass
String	Objref			
String[]	Objref	Array	Object[]	String
&String[]	Objref	Byref Array	Object[]	String
Foo(...,)[]	Objref	Array	Object(...,)[]	Foo
&I4	I4	Byref		
Null	Objref	Null		
&Variant	Valueclass	Byref	Variant	
Variant	Valueclass		Variant	
Variant[]	Objref	Array	Variant[]	Variant
M/R/Boolean	Valueclass		M/R/Boolean	
M/R/Integer4	Valueclass		M/R/Integer4	
&M/R/Byte	Valueclass	Byref	M/R/Byte	

FIG. 5

Refcount (explained later)
Flags (only 1 bit is used here)
StackSlot (number of items on the stack)
Primitive local variable liveness bitmap
All other locals (fixed size array of Item)
Initialised fields bitmap (only present if a value class constructor)
Contents of stack -represented as a variable length array of Items

FIG. 6

"this"	Template	Compatible?
String	Object	Yes
Object	String	No
String[]	Object	Yes
&String	&Object	No
I4	I8	No
M/R/Integer4	I4	Yes
I4	M/R/Integer4	Yes
Object <uninit>	Object	No
Object	Object <uninit>	No
&Object	&Object <addr of local var #5>	No
&Object <addr. of local var #5>	&Object	Yes
&Object	&Object <permanent home>	No
&Object <permanent home>	&Object	Yes
&ValueClass <uninit>	&ValueClass	No

FIG. 7

Item1	Item2	Merge result
String	Object	Object
Object	I4	ERROR
String[]	Object	Object
&String	&Object	DEAD
I4	I8	ERROR
M/R/Integer4	I4	I4
I4	M/R/Integer4	I4
Object <uninit>	Object	DEAD
&ValueClass <uninit>	&ValueClass	&ValueClass <uninit>
&Object	&Object <addr of local var #5>	&Object
&Variant <permanent home>	&Variant	&Variant
InputStream	OutputStream	Stream
InputStream[]	OutputStream[]	Stream[]
Object[][]	Object[]	Array
Object	Null	Object
Foo[(.....)[][][]	Null	Foo[(.....)[][][]
M/R/Integer4[]	Object[]	Array
M/R/Integer1[]	M/R/UnsignedInteger1[]	M/R/Integer1[]

FIG. 8

	I1 U1 Bool VC(I1) VC(U1)	I2 U2 Char VC(I2) VC(U2)	I4 U4 VC(I4) VC(U4)	I8 U8 VC(I8) VC(U8)	R4 VC(R4)	R8 VC(R8)	I U	R	O(I'd)	S(I'y)	S'(I'y)	VC(I'd)
Arg slot	X	X	X	X	X	X	X	X	X	X	X	X
Local slot	X	X	X	X	X	X	X	X	X	X	X	X
Stack slot	-	-	-	X	-	X	X	X	X	X	X	X(C)
Target of &	X	X	X	X	X	X	X	X	X	X	-	X
Field	X	X	X	X	X	X	X	X	X	X	-	X
Argument to ret	X	X	X	X	X	X	X	X	X	-	X(B)	-

FIG. 9(a)

	<i>RefAny</i>	<i>Null</i>	<i>Uninit(ty)</i>	<i>&Uninit(n.ty)</i>	<i>UnconThis</i>	<i>&UninitVCField(n)</i>
Arg slot	X	X	-	X	X	X
Local slot	X	X	X	X	X	X
Stack slot	X	X	-	X	X	X
Target of & Field	-	-	-	-	-	-
Argument to ret	-	X	-	-	-	-

FIG. 9(b)

Type	Purpose
Uninit(<i>ty</i>)	Represents an uninitialized local variable, able to hold data of type <i>ty</i> .
&Uninit(<i>n,ty</i>)	Represents a pointer to the <i>n</i> 'th uninitialized local, which is able to contain data of type <i>ty</i> . Only occurs on the stack prior to initializing a local variable that is a value class.

FIG. 10

Type	Purpose
UnconThis	Represents the uninitialized "this" pointer in a regular class constructor. The "this" pointer, stored in the first argument, becomes available when the constructor calls its superclass constructor, or when construction is delegated to another constructor in the same class.

FIG. 11

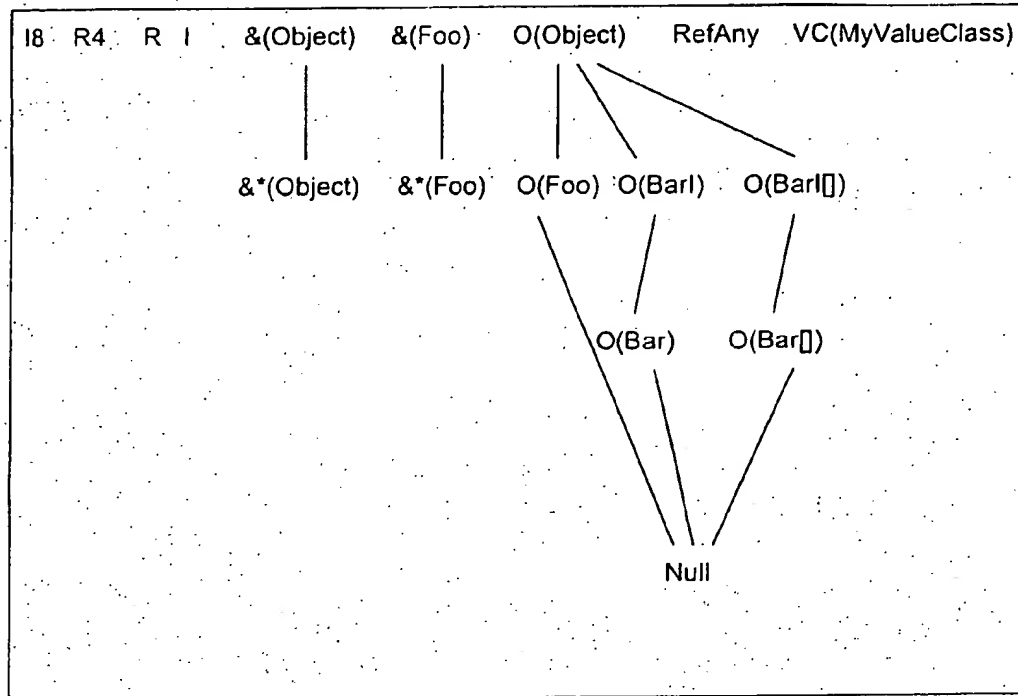
Type	Purpose
UnconThis.	Represents the uninitialized "this" pointer, as in the object constructor case.
&UninitVCField(<i>n,ty</i>)	Represents a pointer to the n'th non-static field of the value class being initialized in a value class constructor, which must be of type <i>ty</i> . These pointers are generated when initializing sub-fields of a value class that are themselves value classes. The initialization is performed by passing the pointer to a constructor.

FIG. 12

Type	Purpose
&*(ty)	Represents a byref with a "home" that will be valid after the stack frame for the method being executed is destroyed.

FIG. 13

FIG. 14



Term	Meaning
<i>address</i>	A number i such that $0 \leq i < \text{length}(\text{codes})$
<i>coerce(ty)</i>	Coercion of a type to a stack type, <ul style="list-style-type: none"> If ty is one of I1, U1, I2, U2, I4, or U4 then the outcome is I. If ty is either I8 or U8 then the outcome is I8. Otherwise the result is ty
$ty1 <: ty2$	The <i>subtype (also called the compatibility)</i> relation. This is the reflexive, transitive closure of the following rules: <ul style="list-style-type: none"> $\text{Null} <: O(td)$. Here td is any class, interface or array type. $O(Intf1) <: O(Intf2)$, if $Intf1$ inherits $Intf2$. Here $Intf1$ and $Intf2$ are interfaces. $\&^*(ty) <: \&(ty)$ $O(C1) <: O(C2)$, if $C1$'s superclass is $C2$ ($C1$ and $C2$ are classes) $O(ty1[]) <: O(ty2[])$, if $O(ty1) <: O(ty2)$ $O(C) <: O(Intf)$, if C implements $Intf$. Here C is a class and $Intf$ is an interface.
$ty1 \cup ty2$	The least upper bound of two types by $ty1 \cup ty2$. This least upper bound may or may not exist – a verification error will arise if we attempt to take the least upper bound of two incompatible types.
$[ty0, \dots, ty_{n-1}]$	An array of types, of length n . We write $[]$ for the empty array.
$\text{array1} + \text{array2}$	Concatenation of two arrays of types.
$[ty1, \dots, ty_m] <: [ty'_1, \dots, ty'_n]$	The subtype relation is extended to arrays of types: let $[ty1, \dots, ty_m] <: [ty'_1, \dots, ty'_n]$ if and only if $m=n$ and $ty_i <: ty'_i$ for $1 \leq i \leq n$.
$[ty1, \dots, ty_m] \cup [ty'_1, \dots, ty'_n]$	The least upper bound of two arrays of types, if it exists.

FIG. 15

Term	Meaning
<i>state</i>	A <i>type state</i> (see above).
<i>state1</i> <: <i>state2</i>	The subtype relation extended to type state. True if and only if <i>args1</i> <: <i>args2</i> , and so on for each corresponding item in the two states. We define that <i>fields1</i> <: <i>fields2</i> iff the flags are pairwise related, with <i>initialised</i> <: <i>uninitialised</i> . We define <i>block1</i> <: <i>block2</i> iff <i>block1</i> and <i>block2</i> are exactly the same.
<i>state1</i> \cup <i>state2</i>	The least upper bound of two type states, if it exists. Note taking the l.u.b. of two <i>fields</i> or two <i>tailcall-ok</i> flags is the same as taking their AND: the outcome is initialised iff both inputs are initialised.
<i>inst</i>	Some IL instruction

FIG. 16

	control flow
beq, bne.un <i>k</i>	<i>next, k</i>
bge <i>k</i> bge.un <i>k</i> bgt <i>k</i> bgt.un <i>k</i> ble <i>k</i> ble.un <i>k</i> blt <i>k</i> blt.un <i>k</i>	<i>next, k</i>
br <i>k</i>	<i>k</i>
brfalse <i>k</i> brtrue <i>k</i>	<i>next, k</i>
endcatch	No destination
endfilter	No destination
endfinally	No destination
jmp <i>tok</i> jmp	Not verifiable
leave <i>k</i>	<i>k</i>
ret	No destination
switch <i>k1, ..., kn</i>	<i>next, k1, ..., kn</i>
call callvirt	No destination if annotated by "tailcall", otherwise like a regular instruction.

FIG. 17

	stack typing	extra conditions
beq, bne.un <i>k</i>	$[ty, ty] \Rightarrow \square$	$ty \in \{I, I8, R, R8\}$ See also the branching restrictions in the IL Instruction Set Description
bge <i>k</i> bge.un <i>k</i> bgt <i>k</i> bgt.un <i>k</i> ble <i>k</i> ble.un <i>k</i> blt <i>k</i> blt.un <i>k</i>	$[ty1, ty2] \Rightarrow \square$	(1) $ty1=ty2$ and $ty1 \in \{I, I8, R\}$, or (2) $ty1, ty2 \in \{I, \&(ty)\}$, or (3) $ty1$ is $O(td1)$ and $ty2$ is $O(td2)$. See also the branching restrictions in the IL Instruction Set Description
br <i>k</i>	$\square \Rightarrow \square$	See the branching restrictions in the IL Instruction Set Description
brfalse <i>k</i> brtrue <i>k</i>	$[I] \Rightarrow \square$	See the branching restrictions in the IL Instruction Set Description
endcatch	$\square \Rightarrow \square$	must be verifying a "catch" section
endfilter	$[I] \Rightarrow \square$	must be verifying a "filter" section.
endfinally	$\square \Rightarrow \square$	must be verifying a "finally" section
jmp <i>tok</i> jmp <i>i</i>	not verifiable	
leave <i>k</i>	$\square \Rightarrow \square$	
ret	$[res] \Rightarrow \square$	The current method's return type is <i>res</i> . A declared return type $\&(ty)$ is treated as $\&*(ty)$ for this purpose, i.e., only byrefs with a home may be returned from methods. Must be verifying a "normal" section.
	$\square \Rightarrow \square$	The current method's return type is void, or it is a constructor. If the method is a constructor, <i>args[0]</i> must not be UnconThis. Furthermore, if the method is a value class constructor then every element of <i>fields</i> must be set, i.e., all value class fields must be initialized. Must be verifying a "normal" section
switch <i>k1,...,kn</i>	$[I] \Rightarrow \square$	See the branching restrictions in the IL Instruction Set Description

FIG. 18

	stack typing	extra conditions
add add.ovf.i add.ovf.u	$[ty, ty] \Rightarrow [ty]$	$ty \in \{I, I8, R\}$
and	$[ty, ty] \Rightarrow [ty]$	$ty \in \{I, I8\}$
arglist	$[] \Rightarrow []$	the method must take a variable number of arguments
ceq	$[ty, ty] \Rightarrow [I]$	$ty \in \{I, I8, R, R8\}$
cgt cgt.un clt clt.un	$[ty1, ty2] \Rightarrow [I]$	(1) $ty1=ty2$ and $ty1 \in \{I, I8, R\}$, or (2) $ty1, ty2 \in \{I, \&(ty)\}$, or (3) $ty1$ is $O(td1)$ and $ty2$ is $O(td2)$.
break	$[] \Rightarrow []$	
ckfinite	$[ty] \Rightarrow [ty]$	$ty \in \{R, R8\}$
conv.t conv.ovf.t	$[ty] \Rightarrow [I]$	$ty \in \{I, I8, R\}$ and $t \in \{I1, U1, I2, U2, I4, U4, I\}$
	$[ty] \Rightarrow [I8]$	$ty \in \{I, I8, R\}$ and $t \in \{I8\}$
	$[ty] \Rightarrow [R]$	$ty \in \{I, I8, R\}$ and $t \in \{R4, R8\}$
cpblk	not verifiable	
div div.un	$[ty, ty] \Rightarrow [ty]$	$ty \in \{I, I8, R\}$
dup	$[ty] \Rightarrow [ty, ty]$	any non-value-class stack type ty
initblk	not verifiable	
jmp tok	not verifiable	
jmp i	not verifiable	
ldarg i	$[] \Rightarrow [\text{coerce}(\text{args}[i])]$	$0 \leq i < \text{NumArgs}$ and $\text{args}[i]$ is not a value class.
ldarga i	$[] \Rightarrow [\&(ty)]$	$0 \leq i < \text{NumArgs}$ and $ty = \text{args}[i]$. ty must be a valid type to construct a $\&$ (see table of types in FIGs. 9(a) and 9(b)) Also set <i>tailcall-ok</i> to false.
ldc.t	$[] \Rightarrow [ty]$	$(t, ty) \in \{(I4, I), (I8, I8), (R4, R), (R8, R)\}$
ldc.i.j	$[] \Rightarrow [I]$	
ldc.i4	$[] \Rightarrow [I]$	
ldftn	not verifiable	
ldind.t	$[\&(ty)] \Rightarrow [\text{coerce}(ty)]$	t is compatible with ty
ldloc i	$[] \Rightarrow [\text{coerce}(\text{locals}[i])]$	$0 \leq i < \text{NumLocals}$ and $\text{locals}[i]$ is not a value class.
ldloca i	$[] \Rightarrow [\&(\text{locals}[i])]$	$0 \leq i < \text{NumLocals}$ and $\text{locals}[i]$ is a valid type to construct a $\&$ (see table of types). Also set <i>tailcall-ok</i> to false.
	$[] \Rightarrow [\&\text{Uninit}(t, ty)]$	$0 \leq i < \text{NumLocals}$ and $\text{locals}[i] = \text{Uninit}(ty)$. This is used to initialize value-class locals. <i>tailcall-ok</i> is unchanged.

FIG. 19(a)

ldnull	$\square \Rightarrow \{\text{Null}\}$	
ldptr	not verifiable	
localloc	not verifiable	
mul mul.ovf.i mul.ovf.u	$\{ty, ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8, R\}$
neg	$\{ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8\}$
nop	$\square \Rightarrow \square$	
not	$\{ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8\}$
or	$\{ty, ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8\}$
pop	$\{ty\} \Rightarrow \square$	any stack type ty , including value class types.
rem	$\{ty, ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8, R\}$
rem.un	$\{ty, ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8\}$
shl, shr, shr.un	$\{ty, \square\} \Rightarrow \{ty\}$	$ty \in \{I, I8\}$
starg i	$\{\text{coerce}(\text{args}[i])\} \Rightarrow \square$	$0 \leq i < \text{NumArgs}$
stind. t	$\{\&(ty), \text{coerce}(ty)\} \Rightarrow \square$	t is compatible with ty
stloc i	$\{\text{coerce}(\text{locals}[i])\} \Rightarrow \square$	This case is used if somewhere in the method there is a $\text{ldloc } i$ and $\text{locals}[i] \neq \text{Uninit}(ty')$, then we must have $0 \leq i < \text{NumLocals}$ and $\text{locals}[i]$ is not a value class type. $\text{locals}[i]$ is unchanged.
	$\{\text{coerce}(ty)\} \Rightarrow \square$	This case is used if somewhere in the method there is a $\text{ldloc } i$ and $\text{locals}[i] = \text{Uninit}(ty)$. We must have $0 \leq i < \text{NumLocals}$ and $\text{locals}[i]$ is not a value class type. $\text{locals}[i]$ is replaced by ty .
	$\{ty\} \Rightarrow \square$	This case is used if somewhere in the method there is a $\text{ldloc } i$. We must have $0 \leq i < \text{NumLocals}$ and $\text{locals}[i]$ is not a value class type. Some l.u.b must exist between ty and $\text{locals}[i]$ (or ty' if $\text{locals}[i]$ is $\text{Uninit}(ty')$), i.e. the base types for ty and the local slot must be compatible. $\text{locals}[i]$ is replaced by ty , i.e. the actual type popped off the stack.
sub sub.ovf.i sub.ovf.u	$\{ty, ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8, R\}$
switch	$\{I\} \Rightarrow \square$	
unaligned	$\square \Rightarrow \square$	
volatile	$\square \Rightarrow \square$	
xor	$\{ty, ty\} \Rightarrow \{ty\}$	$ty \in \{I, I8\}$

FIG. 19(b)

	stack typing	conditions
box tok	$[\&(\text{VC}(td))] \Rightarrow [\text{O}(td)]$	
castclass td	$[\text{O}(td')] \Rightarrow [\text{O}(td)]$	
cpobj td	$[\&(\text{VC}(td)), \&(\text{VC}(td))] \Rightarrow []$ $[\&\text{UnintVCField}(n, \text{VC}(td)), \&(\text{VC}(td))] \Rightarrow []$	In the new type state, <i>fields[n]</i> is set to <i>true</i>
entercrit	$[\text{O}(td)] \Rightarrow []$	
exitcrit	$[\text{O}(td)] \Rightarrow []$	
initobj td	$[\&(\text{VC}(td))] \Rightarrow []$ $[\&\text{UnintVCField}(n, \text{VC}(td))] \Rightarrow []$	In the new type state, <i>fields[n]</i> is set to <i>true</i>
isinst td	$[\text{O}(td')] \Rightarrow []$	
ldelema	$[\text{O}(td), l] \Rightarrow [\&*(ty)]$	<i>td</i> is a 0-based 1-dimensional array of <i>ty</i> 's
ldelem.t	$[\text{O}(td), l] \Rightarrow [\text{coerce}(ty)]$	<i>td</i> is a 0-based 1-dimensional array of <i>ty</i> 's, and <i>t</i> and <i>ty</i> are compatible
ldlen	$[\text{O}(td)] \Rightarrow [l]$	<i>td</i> is a 0-based 1-dimensional array
ldobj td	$[\&(\text{VC}(td))] \Rightarrow [\text{VC}(td)]$	<i>td</i> is a value class. This is used to pass a value class as a parameter.
ldrefany i	$[] \Rightarrow [\text{RefAny}]$	$0 \leq i < \text{NumArgs}$ and the type of argument <i>i</i> is <i>RefAny</i> .
ldrefanya i td	$[] \Rightarrow [\&(\text{O}(td))]$	$0 \leq i < \text{NumArgs}$, <i>td</i> is a typedef token, and the type of argument <i>i</i> is <i>RefAny</i>
ldstr addr	$[] \Rightarrow [\text{O}(td)]$	<i>td</i> is the typedef token for <i>String</i> , and <i>addr</i> points into the .text section, and a null appears at the end of the .text section.
ldtoken tok	$[] \Rightarrow [l]$	<i>tok</i> is a valid metadata token
ldvirtfn tok	not verifiable	
mkrefany td	$[\&(\text{O}(td))] \Rightarrow [\text{RefAny}]$	
newarr.t	$[l] \Rightarrow [\text{O}(td)]$	<i>td</i> refers to a 1-dimensional array of a type compatible with <i>t</i>
newoarr td1	$[l] \Rightarrow [\text{O}(td2)]$	<i>td1</i> refers to a type and <i>td2</i> refers to a 1-dimensional array of <i>td1</i> 's
stelem.t	$[\text{O}(td), l, \text{coerce}(ty)] \Rightarrow []$	<i>td</i> is a 0-based 1-dimensional array of <i>ty</i> 's, and <i>t</i> and <i>ty</i> are compatible.
throw	$[\text{O}(td)]$	
typerefany	not verifiable	
unbox td	$[\text{O}(td)] \Rightarrow [\&(\text{VC}(td))]$	<i>td</i> refers to a value class.

FIG. 20

	stack typing	conditions
call <i>md</i>	$\{coerce(x1), \dots, coerce(xn)\} \Rightarrow []$	<i>md</i> is static, not a constructor, <i>rt</i> = void.
	$\{coerce(x1), \dots, coerce(xn)\} \Rightarrow \{coerce(rt)\}$	<i>md</i> is static, not a constructor, <i>rt</i> is not void.
	$\{T, coerce(x1), \dots, coerce(xn)\} \Rightarrow []$	<i>md</i> is non-static, not a constructor, <i>rt</i> = void.
	$\{T, coerce(x1), \dots, coerce(xn)\} \Rightarrow \{coerce(rt)\}$	<i>md</i> is non-static, not a constructor, <i>rt</i> is not void.
	$\{ty, coerce(x1), \dots, coerce(xn)\} \Rightarrow []$	<i>md</i> is a non-static, non-value-class constructor.
		<p>We must be verifying (as well as calling) a constructor, thus we are calling our superclass constructor, or delegating construction to another constructor in our own class. After we do this the "this" pointer (stored in <i>args[0]</i>) becomes available for normal use. More formally, this means:</p> <ul style="list-style-type: none"> • $T = O(this)$ or $T = O(super)$, where <i>this</i> is the type containing the method being verified and <i>super</i> is its superclass, • $ty = UnconThis$, and • $args[0] = UnconThis$ <p>In the new type state <i>args[0]</i> now becomes $O(this)$.</p>
	$\{ty, coerce(x1), \dots, coerce(xn)\} \Rightarrow []$	<p><i>md</i> is a non-static value-class constructor.</p> <p>Three possibilities arise:</p> <p>(1) We are initializing a local variable that is a value class, by calling a constructor on the appropriate pointer. That is, $ty = \&Uninit(n, VC(T))$, and in the new type state <i>locals[n]</i> is set to $VC(T)$, or</p> <p>(2) We are verifying (as well as calling) a value-class-constructor, and we are initializing a value class field within the value class being constructed. That is, $ty = \&UninitVCField(n, VC(T))$. In the new state we update <i>fields[n]</i> to true, so that field is now considered initialized. If all of <i>fields</i> is now set, and <i>args[0]</i> contains <i>UnconThis</i>, then <i>args[0]</i> becomes $\&VC(this)$ in the new state.</p> <p>(3) We are delegating construction as above, by calling another constructor in our own class. After we do this the "this" pointer (stored in <i>args[0]</i>) becomes available for normal use. More formally, this means:</p> <ul style="list-style-type: none"> • <i>T</i> must be the type containing the method being verified, • $args[0] = ty = UnconThis$ <p>In the new type state <i>args[0]</i> becomes $\&VC(this)$.</p>

FIG. 21(a)

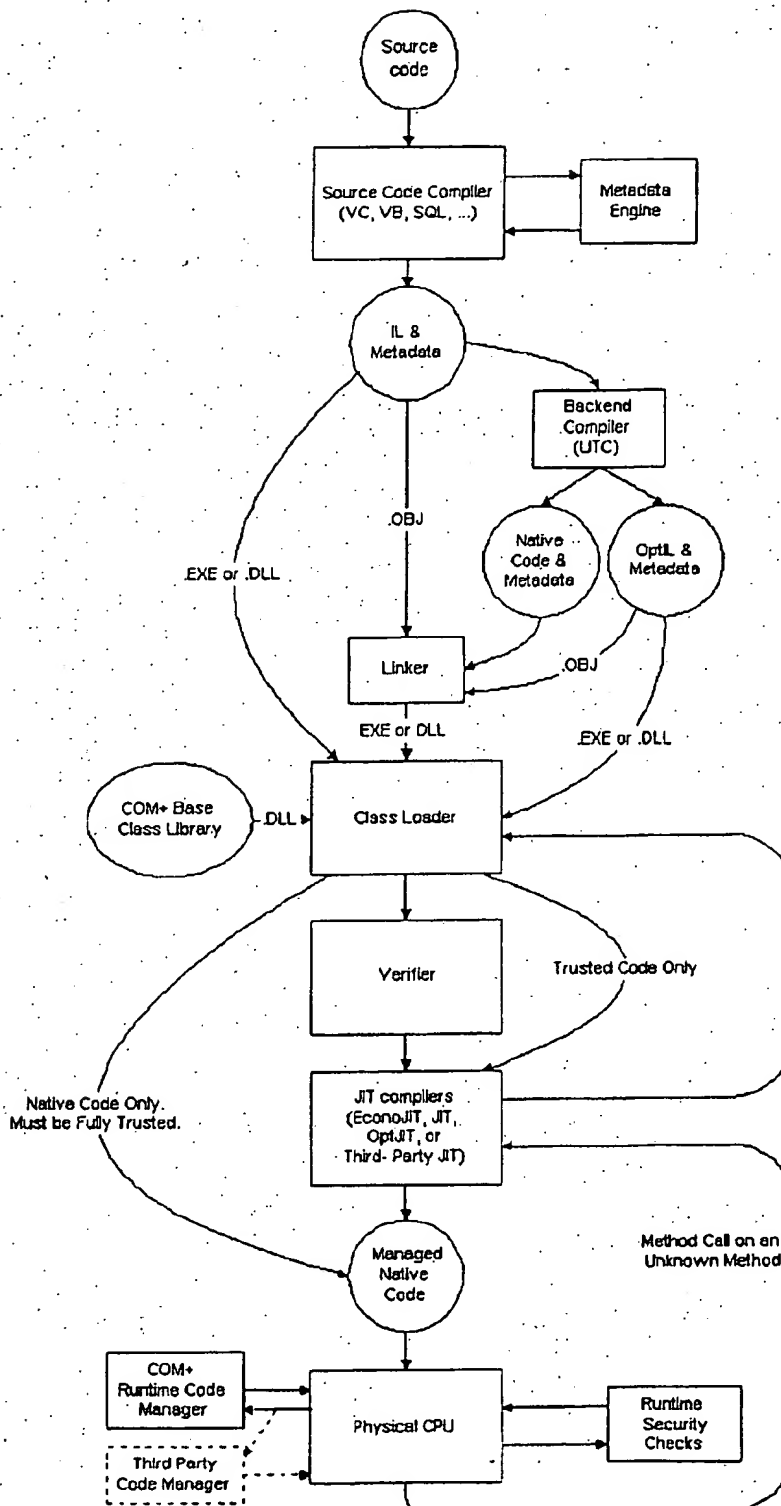
		In the above cases where <i>rt</i> = void, then if the instruction is annotated by <i>tailcall</i> then the current method's return type must be void, and <i>tailcall-ok</i> must be true.
		In the above cases where <i>rt</i> is not void, then if the instruction is annotated by <i>tailcall</i> then <i>rt</i> <: <i>res</i> where the current method's return type is <i>res</i> and <i>tailcall-ok</i> must be true.
<i>calli call/siledescr</i>	not verifiable	
<i>callvirt md</i>	$\{T, \text{coerce}(x1), \dots, \text{coerce}(xn)\} \Rightarrow []$	<i>md</i> is virtual, <i>rt</i> = void. If the instruction is annotated by <i>tailcall</i> then the current method's return type must be void, and <i>tailcall-ok</i> must be true.
	$\{T, \text{coerce}(x1), \dots, \text{coerce}(xn)\} \Rightarrow [\text{coerce}(rt)]$	<i>md</i> is virtual, <i>rt</i> is not void. If the instruction is annotated by <i>tailcall</i> then <i>rt</i> <: <i>res</i> where the current method's return type is <i>res</i> and <i>tailcall-ok</i> must be true.
<i>newobj md</i>	$[\text{coerce}(x1), \dots, \text{coerce}(xn)] \Rightarrow [O(T)]$	<i>md</i> is a non-static constructor, and <i>md</i> resolves to a method with signature <i>rt</i> <i>T::method</i> (<i>x1</i> , ..., <i>xn</i>) where <i>rt</i> is the return type.

FIG. 21(b)

	stack typing	conditions
ldfld <i>fd</i>	$[O(T)] \Rightarrow [\text{coerce}(fy)]$ $[\&(VC(T))] \Rightarrow [\text{coerce}(fy)]$	
ldflda <i>fd</i>	$[O(T)] \Rightarrow [\&^*(fy)]$ $[\&(VC(T))] \Rightarrow [\&(fy)]$ $[\&^*(VC(T))] \Rightarrow [\&^*(fy)]$ $[\text{UnconThis}] \Rightarrow [\&\text{UninitVCField}(n, fy)]$	$T = \text{this}$ where <i>this</i> is the type of the method being verified, which must be a value class. <i>n</i> must be the field index of the field within <i>T</i> .
ldsfld <i>fd</i>	$[] \Rightarrow [\text{coerce}(fy)]$	
ldsflda <i>fd</i>	$[] \Rightarrow [\&^*(fy)]$	
stfld <i>fd</i>	$[O(T), \text{coerce}(fy)] \Rightarrow []$ $[\&VC(fy), \text{coerce}(fy)] \Rightarrow []$ $[\text{UnconThis}, \text{coerce}(fy)] \Rightarrow []$	$T = \text{this}$ where <i>this</i> is the type of the method being verified. <i>T</i> must be a value class. If <i>n</i> is the index of the field in <i>T</i> then <i>fields[n]</i> becomes set in the new type state, i.e., the field becomes initialized. If all of <i>fields</i> is now set, and <i>args[0]</i> contains <i>UnconThis</i> , then <i>args[0]</i> becomes $\&(VC(\text{this}))$ in the new state.
stsfld <i>fd</i>	$[\text{coerce}(fy)] \Rightarrow []$	

FIG. 22

FIG.
23



I1	8-bit 2's complement signed value
U1	8-bit unsigned binary value
I2	16-bit 2's complement signed value
U2	16-bit unsigned binary value
I4	32-bit 2's complement signed value
U4	32-bit unsigned binary value
I8	64-bit 2's complement signed value
U8	64-bit unsigned binary value
R4	32-bit IEEE 754 floating point value
R8	64-bit IEEE 754 floating point value
I	natural size 2's complement signed value
U	natural size unsigned binary value, also unmanaged pointer
R	natural size floating point value
O	natural size object reference to managed memory
&	natural size managed pointer (may point into managed memory)

FIG. 24

add	Numeric
add.ovf.*	Specific Integer
and	Integer
beq[.s]	Numeric
bge[.s]	Numeric
bge.un[.s]	Unsigned/unordered
bgt[.s]	Numeric
bgt.un[.s]	Unsigned/unordered
ble[.s]	Numeric
ble.un[.s]	Unsigned/unordered
blt[.s]	Numeric
blt.un[.s]	Unsigned/unordered
bne.un[.s]	Unsigned/unordered
ceq	Numeric
cgt	Numeric
cgt.un	Unsigned/unordered
ckfinite	Floating point
clt	Numeric
clt.un	Unsigned/unordered
conv.*	Specific
conv.ovf.*	Specific

div	Numeric
div.un	Integer unsigned
ldc.*	Load Constant
ldelem.*	Specific
ldind.*	Specific
mul	Numeric
mul.ovf.*	Specific
neg	Integer
newarr.*	Specific
not	Integer
or	Integer
rem	Numeric
rem.un	Integer unsigned
shl	Integer
shr	Integer
shr.un	Specific
stelem.*	Specific
stind.*	Specific
sub	Numeric
sub.ovf.*	Specific
xor	Integer

FIG. 25

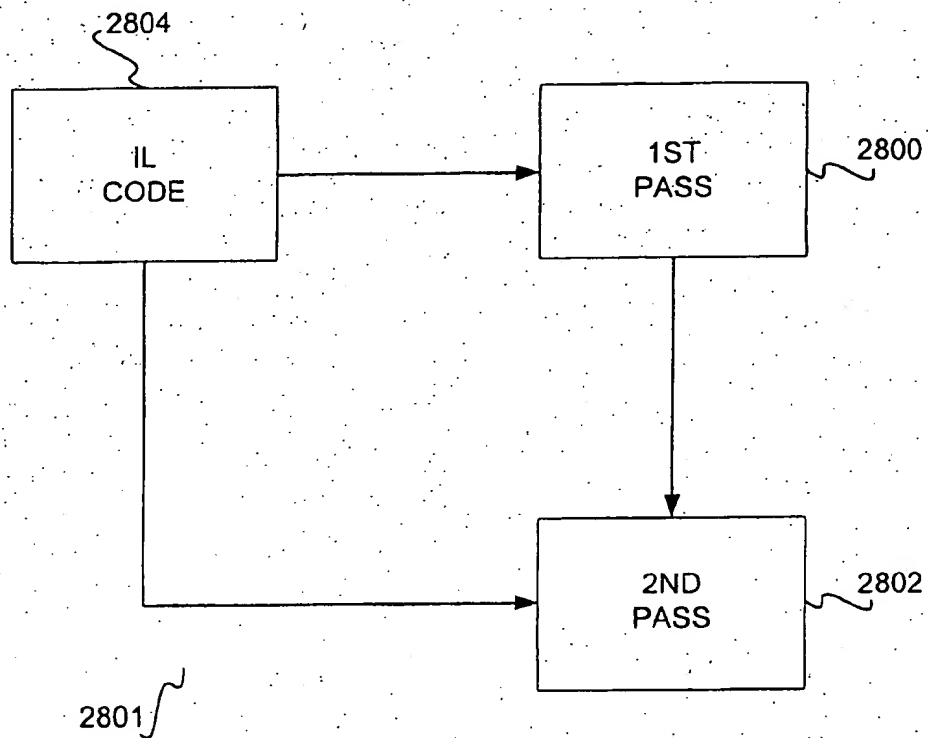
Type of Home	Runtime Address Computation	JITtime Type Determination
Argument	ldarga	Method signature
Local Variable	ldloca	Locals signature in method header
Field	ldlfda	Type of object
Static	ldsifda (or ldflfa)	Type of class, interface, or module
Array Element	ldelema (or library call)	Type of array

FIG. 26

Type of data	Pass By	How data is sent
Built-in value type (int, float, etc.)	Value	Copied to called method, type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Ref-Any	Address sent along with type information to called method
User-defined value class	Value	Called method receives a copy (but copy may be done by either calling method or called method); type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Ref-Any	Address sent along with type information to called method
Object class	Value	Reference to data sent to called method, type statically known and class available from reference
	Reference	Address of reference sent to called method, type statically known and class available from reference
	Ref-Any	Address of reference sent to called method along with static type information, class (i.e. dynamic type) available from reference

FIG. 27

FIG 28



VERIFIER TO CHECK INTERMEDIATE LANGUAGE

FIELD OF THE INVENTION

[0001] This invention relates generally to intermediate languages, such as intermediate languages compilable from multiple language sources, and more particularly a verifier for such intermediate language.

BACKGROUND OF THE INVENTION

[0002] Intermediate language-type models for programming languages have become increasingly popular. In an intermediate language model, a source code is generally compiled into a desirably substantially platform independent intermediate language. When the code is desired to be run on a particular platform, an execution engine on that platform then just-in-time compiles, interprets, or compiles the intermediate language to native code understandable by the platform. Examples of intermediate language-type computer programming languages include Component Object Model Plus (COM+), and the Java programming language.

[0003] A difficulty arises in intermediate language-type models in that, in some circumstances, the execution engine needs to run untrusted code. Untrusted code is code having an author that cannot be authenticated, and thus may cause problems with the host machine running the code. For example, untrusted code could attack the execution engine. This could be accomplished by, for example, casting between integers and object references, accessing fields that are intended to be private, failing to initialize objects correctly, overflowing the stack, or referring to arguments or local variables that do not exist.

[0004] One solution to this problem is to construct a defensive execution engine that performs a variety of run-time checks to prevent all such attacks. However, this can cause a substantial reduction in execution speed for untrusted code, rendering the execution engine an inefficient platform.

[0005] For these and other reasons, there is a need for the invention.

SUMMARY OF THE INVENTION

[0006] The invention relates to a verifier to check intermediate language code. In one embodiment, a computer-implemented method first verifies metadata of an intermediate language code for consistency and accuracy, and then verifying the intermediate language code for consistency and accuracy. This latter part in one embodiment is accomplished by performing first a syntactic check of the intermediate language code, and then a semantic check of the intermediate language code.

[0007] Embodiments of the invention thus provide for advantages not found in the prior art. Static type-checking can be performed on untrusted intermediate language code, such that only well-typed untrusted code is executed. Type-checking is performed by a verifier in one embodiment. The type system implemented by the verifier can be designed to prevent attacks on the execution engine by untrusted execution. The type system can check and verify byrefs, value classes, refany's, and native-size primitive types, among other aspects.

[0008] The invention includes computer-implemented methods, machine-readable media, computerized systems, and computers of varying scopes. Other aspects, embodiments and advantages of the invention, beyond those described here, will become apparent by reading the detailed description and with reference to the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] FIG. 1 is a diagram of an operating environment in conjunction with which embodiments of the invention can be practiced;

[0010] FIG. 2 is a diagram of a system according to an embodiment of the invention;

[0011] FIG. 3 is a flowchart of a method according to an embodiment of the invention;

[0012] FIG. 4 is a table of flags used by the verifier in one embodiment of the invention;

[0013] FIG. 5 is a table listing example flags used by the verifier in one embodiment of the invention;

[0014] FIG. 6 is a layout of an EntryState data structure that is used by the verifier in one embodiment of the invention;

[0015] FIG. 7 is a table showing the functionality of the item CompatibleWith that is used by the verifier in one embodiment of the invention;

[0016] FIG. 8 is a table showing examples of merging that is used by the verifier in one embodiment of the invention;

[0017] FIG. 9(a) and FIG. 9(b) show a table of different types that appear at different places during verification, according to one embodiment of the invention;

[0018] FIG. 10 is a table showing special types for verifying the correct use of uninitialized locals, according to one embodiment of the invention;

[0019] FIG. 11 is a table summarizing the IL type verification that constructors must construct, according to one embodiment of the invention;

[0020] FIG. 12 is a table summarizing the IL type verification that value class fields must be initialized, according to one embodiment of the invention;

[0021] FIG. 13 is a table summarizing the IL type verification that only special Byref's may be returned, according to an embodiment of the invention;

[0022] FIG. 14 is a diagram showing an example illustrating the IL type verification of compatibility for types according to an embodiment of the invention;

[0023] FIG. 15 is a diagram showing the definitions for the semantic checks of the second pass of IL verification, according to one embodiment of the invention;

[0024] FIG. 16 is a table summarizing the IL verification of semantic checks, specifically type states, according to an embodiment of the invention;

[0025] FIG. 17 is a table specifying the destination addresses of an instruction, for use in the IL verification of semantic checks, according to an embodiment of the invention;

[0026] FIG. 18 is a table showing the effect of various control flow instructions on the type state, in conjunction with the IL verification of semantic checks, according to an embodiment of the invention;

[0027] FIG. 19(a) and FIG. 19(b) are a table showing the effect of various basic flow instructions on the type state, in conjunction with the IL verification of semantic checks, according to an embodiment of the invention;

[0028] FIG. 20 is a table showing the effect of various object model instructions on the type state, in conjunction with the IL verification of semantic checks, according to an embodiment of the invention;

[0029] FIG. 21(a) and FIG. 21(b) are a table showing the effect of various method call instructions on the type state, in conjunction with the IL verification of semantic checks, according to an embodiment of the invention;

[0030] FIG. 22 is a table showing the effect of various field instructions on the type state, in conjunction with the IL verification of semantic checks, according to an embodiment of the invention;

[0031] FIG. 23 is a diagram of a virtual execution system according to an embodiment of the invention;

[0032] FIG. 24 is a table showing the data types directly supported by an execution engine in one embodiment of the invention;

[0033] FIG. 25 is a table showing the intermediate language instructions that deal with numeric values, along with the category to which they belong, according to one embodiment of the invention;

[0034] FIG. 26 is a table summarizing a manner to compute the address of the home location and a manner to determine the type of a home location, according to one embodiment of the invention;

[0035] FIG. 27 is a table the parameter passing convention used for each data type, according to one embodiment of the invention; and,

[0036] FIG. 28 is a diagram of an implementation of a verifier according to an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0037] In the following detailed description of exemplary embodiments of the invention, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration specific exemplary embodiments in which the invention may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention, and it is to be understood that other embodiments may be utilized and that logical, mechanical, electrical and other changes may be made without departing from the spirit or scope of the present invention. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present invention is defined only by the appended claims.

[0038] Some portions of the detailed descriptions which follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer

memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated.

[0039] It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as processing or computing or calculating or determining or displaying or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[0040] Operating Environment

[0041] Referring to FIG. 1, a diagram of the hardware and operating environment in conjunction with which embodiments of the invention may be practiced is shown. The description of FIG. 1 is intended to provide a brief, general description of suitable computer hardware and a suitable computing environment in conjunction with which the invention may be implemented. Although not required, the invention is described in the general context of computer-executable instructions, such as program modules, being executed by a computer, such as a personal computer. Generally, program modules include routines, programs, objects, components, data structures, etc., that perform particular tasks or implement particular abstract data types.

[0042] Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, network PC's, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

[0043] The exemplary hardware and operating environment of FIG. 1 for implementing the invention includes a general purpose computing device in the form of a computer 20, including a processing unit 21, a system memory 22, and a system bus 23 that operatively couples various system components include the system memory to the processing unit 21. There may be only one or there may be more than one processing unit 21, such that the processor of computer 20 comprises a single central-processing unit (CPU), or a

plurality of processing units, commonly referred to as a parallel processing environment. The computer 20 may be a conventional computer, a distributed computer, or any other type of computer; the invention is not so limited.

[0044] The system bus 23 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. The system memory may also be referred to as simply the memory, and includes read only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system (BIOS) 26, containing the basic routines that help to transfer information between elements within the computer 20, such as during start-up, is stored in ROM 24. The computer 20 further includes a hard disk drive 27 for reading from and writing to a hard disk, not shown, a magnetic disk drive 28 for reading from or writing to a removable magnetic disk 29, and an optical disk drive 30 for reading from or writing to a removable optical disk 31 such as a CD ROM or other optical media.

[0045] The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical disk drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules and other data for the computer 20. It should be appreciated by those skilled in the art that any type of computer-readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read only memories (ROMs), and the like, may be used in the exemplary operating environment.

[0046] A number of program modules may be stored on the hard disk, magnetic disk 29, optical disk 31, ROM 24, or RAM 25, including an operating system 35, one or more application programs 36, other program modules 37, and program data 38. A user may enter commands and information into the personal computer 20 through input devices such as a keyboard 40 and pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port, or a universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor, computers typically include other peripheral output devices (not shown), such as speakers and printers.

[0047] The computer 20 may operate in a networked environment using logical connections to one or more remote computers, such as remote computer 49. These logical connections are achieved by a communication device coupled to or a part of the computer 20; the invention is not limited to a particular type of communications device. The remote computer 49 may be another computer, a server, a router, a network PC, a client, a peer device or other common network node, and typically includes many or all of the elements described above relative to the computer 20, although only a memory storage device 50 has been illus-

trated in FIG. 1. The logical connections depicted in FIG. 1 include a local-area network (LAN) 51 and a wide-area network (WAN) 52. Such networking environments are commonplace in office networks, enterprise-wide computer networks, intranets and the Internet, which are all types of networks.

[0048] When used in a LAN-networking environment, the computer 20 is connected to the local network 51 through a network interface or adapter 53, which is one type of communications device. When used in a WAN-networking environment, the computer 20 typically includes a modem 54, a type of communications device, or any other type of communications device for establishing communications over the wide area network 52, such as the Internet. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the personal computer 20, or portions thereof, may be stored in the remote memory storage device. It is appreciated that the network connections shown are exemplary and other means of and communications devices for establishing a communications link between the computers may be used.

[0049] Overview

[0050] In this section, description is provided of an overview of an embodiment of the invention. The overview is a general description of one embodiment of the invention. Both a description of a system and a method, according to embodiments of the invention, are provided. After this section, a more detailed description of one embodiment of the invention is provided, with specific reference to the COM+ programming language, that is, the intermediate language supported by the COM+ run-time. Those of ordinary skill within the art can appreciate, however, that the invention is not so limited to COM+ or any other intermediate language-type programming language; one other example of such a programming language is the Java programming language, that is, the byte code language of the Java virtual machine.

[0051] Referring first to FIG. 2, a diagram of a system according to an embodiment of the invention is shown. The system includes an execution engine 200, of which a verifier 202 is a part. Multiple source code language sources, such as Visual Basic (VB), Visual C++ (VC++), and other sources, are compiled, by compilers such as compilers 204, into intermediate language (IL) code. The execution engine 200 then compiles, interprets, or just-in-time compiles the IL code into executable code (based on the intermediate code) native to a particular platform. That is, the execution engine 200 translates the intermediate code to the executable code for execution. The verifier 202 verifies the IL code prior to the code being compiled, interpreted, or just-in-time compiled (that is, prior to translation thereof to executable code).

[0052] In one embodiment, the verifier 202 verifies the intermediate language code as well as metadata of the IL code (as will be described). Furthermore, in one embodiment, the verifier 202 verifies the IL code by performing a syntactic and a semantic check of the IL code, as will be described later in the detailed description. For example, a semantic check of the code can in varying embodiments include checking byrefs and refany's of the code (referred to generally as reference aspects of the code), value classes of the code, native-size primitive types of the code, tail calls of

the code, as well as other aspects of the IL code. The invention is not so limited, however.

[0053] Each of the compilers 204, the verifier 202, and the execution engine 200 can in one embodiment be a computer program executed by a processor from a computer-readable medium such as a memory. Compilers such as compilers 204 are known within the art. The verifier 202 and the execution engine 200 in one embodiment is as is described in more detail later in the detailed description. Each of the multiple source code language source, the IL code, and the executable code can in one embodiment be stored as data on a computer-readable medium, such as a memory or a hard disk drive. The invention is not so limited, however.

[0054] Referring next to FIG. 3, a flowchart of a computer-implemented method according to an embodiment of the invention is shown. The method of FIG. 3 in one embodiment is the manner by which the verifier 202 of FIG. 2 accomplishes verification of IL code, although the invention is not necessarily so limited. The computer-implemented method is desirably realized at least in part as one or more programs running on a computer—that is, as a program executed from a computer-readable medium such as a memory by a processor of a computer. The programs are desirably storable on a machine-readable medium such as a floppy disk or a CD-ROM, for distribution and installation and execution on another computer.

[0055] In 300, the method first performs verification of metadata of the IL code for consistency and accuracy. Metadata of the IL code is described in more detail later in the detailed description. In 302, the method next performs verification of the IL code itself for consistency and accuracy. This is conducted in one embodiment in two passes: a first pass 304 to perform syntactic checks of the IL code, and a second pass 306 to perform semantic checks of the IL code. Example syntactic and semantic checks are described later in the detailed description. However, semantic checks in varying embodiments of the invention include: reference aspect checks (such as byref and refany checks), value class checks, native-size primitive type checks, and tail call verifications, as are described in more detail later in the detailed description.

[0056] Not shown in FIG. 3 is that the method can include initially generating the intermediate language code by compiling a source code, such as a source VC++ or VB code, into the IL code. Furthermore, not shown in FIG. 3 is that the method can include subsequently executing the intermediate language code upon verification of the code and the metadata thereof, by translating the intermediate language code into executable code native to a host platform (i.e., by compiling, just-in-time compiling, or interpreting the IL code into executable code). The invention is not so limited, however.

[0057] Implementation of an IL Verifier

[0058] In this section of the detailed description, a general implementation of an IL verifier, according to one embodiment of the invention, is presented. A more specific implementation of a verifier is presented in the next sections of the detailed description. The implementation described in this section is made with reference to FIG. 28.

[0059] Referring now to FIG. 28, a verifier 2801 according to an embodiment of the invention includes a first-pass

component 2800 and a second-pass component 2802. Each component 2800 and 2802 can in one embodiment be implemented as a computer program, or one or more parts thereof, executed by a processor from a computer-readable medium such as a memory or a hard disk drive. The IL code 2804 to be verified is input into both components 2800 and 2802. Generally, the component 2800 performs syntactic checks, and the component 2802 performs semantic checks, as has been described.

[0060] In the first pass conducted by the component 2800, the locations of all the basic blocks of the code 2804 are determined, and in one embodiment are prepared into a sorted list. The basic blocks of the code 2804 are defined in one embodiment as minute sections of the IL code 2804 such that each basic block includes one or more instructions of the IL code 2804. The first pass ensures that all branch destinations—for example, jump instructions to other basic blocks of code—are proper, and in one embodiment, ensures that all branch destinations are on instruction boundaries, as opposed to within the middle of an instruction, for example.

[0061] In the second pass conducted by the component 2802, other checks are performed, as described within the next sections of the detailed description. For each opcode to be verified, in particular, there is a list of operations and checks that can be performed. In one embodiment, for most opcodes, this means that no code needs to be written to handle the opcode; instead, a string exists for the opcode that details how the verification should be performed. For example, a string such as “44:4” may specify “pop two four byte integers from the stack, and then push a four byte integer on the stack.”

[0062] In one particular embodiment, for aid in verification, some assumptions are made regarding the code. First, the type of the arguments to any method in the code being verified are fixed. If an argument is declared as a string, for example, then whenever the argument is loaded onto the stack, a string will always result, even if a subclass of a string is stored. This means that dynamic type information for arguments do not need to be maintained on a per-basic block basis.

[0063] Second, the type of primitive local variables, such as integers, floating points, etc., is fixed. If a variable is declared as being a primitive type, then it is not allowed to store a non-primitive type, such as an object reference, into it. In this manner, a single bit—“live” or “dead” for example—is sufficient to convey a primitive variable’s state at any point. For non-primitive variables, however, complete type information must be maintained.

[0064] More Specific Implementation Design of an IL Verifier

[0065] The next three sections of the detailed description, including this section, relate to varying degrees specifically to the COM+ intermediate language—that is, to a verifier and an execution engine for the COM+ intermediate language. The description provided in these sections is such that it is understandable to those of ordinary skill within the art. It is noted, however, that the invention itself is not limited to the details provided herein in these three sections. In this section in particular, the implementation design of an IL verifier, as can be used in conjunction with a system or a method of an embodiment of the invention as already described, is provided.

[0066] Background

[0067] The verifier in one embodiment simulates method execution by pushing and popping the appropriate types as instructions are seen, as can be appreciated by those of ordinary skill within the art. Loads from and stores into local variables, arguments, and fields are checked for type safety, as are the arguments to method calls.

[0068] The verifier keeps track of the "current state" as it simulates instructions. The current state includes the contents of the stack, the local variables, and the arguments. If verifying a constructor method, it also keeps track of the initialization status of the "this" pointer.

[0069] Whenever execution leaves the current basic block and can enter one or more basic blocks, the current state is propagated to all unvisited destination basic blocks, which are then queued for visiting. Destination basic blocks which have already been visited will already have a state, E, associated with them; this state is compared with the current state, S. If S is compatible with E (i.e. S is a subset of E), then nothing happens. If S is not a subset of E, then the S and E are merged conservatively, and this merged state is stored back into the destination basic block, which is then queued for re-visiting.

[0070] A new basic block is dequeued whenever the verifier encounters a control flow instruction which does not cause it to visit other basic blocks (for example, a "ret" instruction, or a conditional branch where all destinations have already been visited, and the states match). Otherwise it follows one of the unvisited control flow paths.

[0071] Two Pass System

[0072] The verifier uses a two pass system to verify a method. The first pass is simple and fast, and performs the following tasks:

[0073] It determines the location of all of the basic blocks, and creates a sorted array of them (m_pBasicBlockList).

[0074] It ensures that all jump destinations land on instruction boundaries, and are inside the method.

[0075] It determines which locals have their address taken. This is because if a local declared to contain an objref has its address taken, then its type is fixed—no matter what is stored into the local, reading from the local will always return its declared type (due to the problem of aliasing, the verifier must always be conservative here).

[0076] A new basic block starts on:

[0077] The destination of a conditional branch

[0078] The instruction after a conditional branch, unconditional branch, throw, or return

[0079] All destinations of a switch, including the fall through case

[0080] The start of a try clause

[0081] The end of a try clause (i.e. the instruction after the last instruction inside the try clause)

[0082] The start of a catch clause handler

[0083] The start of a filter clause handler (if the exception clause has a filter)

[0084] All three tasks are accomplished by creating two bitmaps. These bitmaps are short lived, and exist only on this quick first pass of the verifier. Each bitmap has one bit per byte of IL code. The first bitmap will have a bit set corresponding to the start of each instruction, while the second bitmap will have a bit set corresponding to the start of each basic block.

[0085] First, the exception list is traversed, and the catch clauses, filters, and start and end of try clauses are set as being the start of a basic block. The first instruction in the method is also set as being the start of a basic block.

[0086] Then, the method's instructions are decoded from start to end in a linear fashion, not following control flow. A safe function is used to conservatively decode opcodes and operand data, which detects invalid instructions, or running off the end of the code.

[0087] The appropriate bits in the start-of-instruction bitmap are set as the opcodes are decoded. As control flow instructions are seen (all branches, throw, return, switch), the appropriate bits are set in the basic block boundary bitmap.

[0088] Once the end of the method is reached, a check is made to ensure that wherever a basic block was said to begin, it was also the beginning of an instruction. This catches attempts to jump into the middle of an instruction.

[0089] A sorted list of basic blocks is made by traversing the basic block bitmap linearly.

[0090] Data Structures

[0091] There are Three Significant Data Structures in the Verifier:

[0092] Item. An Item represents a data type, and is used to store the current contents of local variables, stack entries, and globally known argument types and local variable types.

[0093] BasicBlock. A basic block contains only two fields—the program counter (offset into the method code) where the basic block starts, and an EntryState, providing the logical state of the machine at that point (or NULL if the basic block has not been visited or queued yet).

[0094] EntryState. Provides a description of the current state. Defines the contents and liveness of all local variables, the size and contents of the stack, whether the "this" pointer is initialized. If verifying a value class constructor method, there is also a bitmap indicating which fields have been initialized.

[0095] An Item comprises three components:

[0096] m_dwFlags (a DWORD containing a variety of information)

[0097] m_pEEClass (optional)

[0098] m_pUnderlyingArrayClass (optional)

[0099] The lower 6 bits of m_dwFlags provide basic type information (known as the "type component"), as defined in CorElementType. Only a subset of types in

[0100] CorElementType are used:

[0101] ELEMENT_TYPE_I1

[0102] ELEMENT_TYPE_I2

[0103] ELEMENT_TYPE_I4
 [0104] ELEMENT_TYPE_I8
 [0105] ELEMENT_TYPE_R4
 [0106] ELEMENT_TYPE_R8
 [0107] ELEMENT_TYPE_CHAR
 [0108] ELEMENT_TYPE_BOOLEAN
 [0109] ELEMENT_TYPE_VOID

[0110] In addition to these types, the verifier extends the enumeration with the following:

[0111] VER_ELEMENT_TYPE_OBJREF
 [0112] VER_ELEMENT_TYPE_BYREF
 [0113] VER_ELEMENT_TYPE_VALUE_CLASS

[0114] The CorElementType enumerations for ELEMENT_TYPE_CLASS and ELEMENT_TYPE_VALUE_CLASS are not used. Unsigned types (such as ELEMENT_TYPE_U4) never appear anywhere in the verifier—they are always converted to their equivalent signed type. Naked ELEMENT_TYPE_I1/I2/Char/Boolean types cannot exist on the stack, in one embodiment. The verifier promotes small integral values to I4's when moving them onto the stack (for example, from local variables). However, it is possible to have a byref of one of these types (for example, a "&I2"). If the item is an object reference, or a byref object reference, the type component will be VER_ELEMENT_TYPE_OBJREF.

[0115] In addition to the type component (lower 6 bits of m_dwFlags), the flags as shown in the table of FIG. 4 reside in the upper bits of m_dwFlags. Some example flags are shown in the table of FIG. 5. There is a duality between some value classes and their primitive types, which can be disadvantageous. For example, a Microsoft/Runtime/Integer4 value class is really equivalent to an ELEMENT_TYPE_I4, even though the metadata signatures are different, and they have completely different representations in the verifier. Both forms can exist throughout the EE.

[0116] The verifier handles this issue by ensuring that at the lowest level, the CompatibleWith and Merge functions are aware of this duality. The rest of the verifier is not aware of the duality. Therefore the CompatibleWith function should be used to determine compatibility between two Items, rather than peeking at the contents of Items directly. In other words, instead of checking "Item.GetType()=ELEMENT_TYPE_I4", create an Item2 which is an ELEMENT_TYPE_I4, and then call "Item.CompatibleWith(&Item2)". It is noted that the Item.GetType() helper function does not actually mask off and return the type component directly. If has an additional check—if the item is a byref, it returns ELEMENT_TYPE_BYREF instead.

[0117] Referring next to an EntryState, an EntryState is a variable size data structure, describing the machine state at a particular basic block. There is a fixed size portion, which is fixed when verifying a given method (but different for different methods), the size of which is determined by the number of local variables, which locals have fixed types and require only live/dead information, and whether this is a value class constructor. The variable size portion depends on

the number of items on the stack. This tends to be zero most of the time (basic block boundaries tend to have an empty stack).

[0118] The layout for an entry state is shown in FIG. 6. The variable Verifier::m_StackItemOffset provides the offset at which the stack contents begin (i.e. the size of the fixed portion of the EntryState). Thus, the size of an EntryState is Verifier::m_StackItemOffset+sizeof(Item)*EntryState.m_StackSlot. All locals which are primitive (i.e. have a negative slot number) have their liveness stored as a single bit in the primitive local variable liveness bitmap. All other locals (those with slot numbers>=0) have an Item allocated for them in the EntryState.

[0119] The EntryState structure is refcounted to conserve memory. Consider a situation where there is a 20-way switch statement (for simplicity, assume all destinations are unvisited). Rather than allocating 20 separate but identical EntryStates and attaching them to the destination basic blocks, the verifier instead allocates one EntryState, gives it a refcount of 20, and points the EntryState of each of the destination basic blocks at it. It is rare to ever have to re-store a new EntryState into a basic block, since basic blocks are almost never revisited for most code.

[0120] If for some reason one of these basic blocks, B, must be revisited in the future (due to a compatibility conflict), then the verifier will not modify the original EntryState, but will instead make a new copy and make B point to that instead. Refcounting occurs even for two way control flow (e.g. condition branches). Since it is quite rare to revisit a basic block, this is advantageous.

[0121] Revisiting

[0122] Revisiting is a quite rare occurrence. An example shows when it occurs:

```
[0123] A:
      [0124] Ldarg 0
      [0125] Ldc.i45
      [0126] Bne C
[0127] B:
      [0128] ldc.i499
      [0129] Stloc 3
      [0130] Push InputStream
      [0131] Goto X
[0132] C:
      [0133] Push OutputStream
      [0134] Goto X
[0135] X:
      [0136] Call Stream::SomeFunction()
      [0137] Return
```

[0138] The verifier starts simulating at the beginning of A. At the end of A, when it sees the "Bne C" instruction, it queues basic block C for visiting with EntryState "stack="

{}", and falls through to execute the B case. At the end of B it notices X is unvisited, so it jumps to X, and stores "loc#3=live, stack={InputStream}" in X's EntryState. After the end of X, control flow ends, so it dequeues a basic block. Since C is the only basic block in the queue, C is dequeued (with a state of stack={}). At the end of "C", when the current state is "stack={OutputStream}", it notices that X is already visited, so it compares its current state with X's stored state (loc#3=live, stack={InputStream}). The current state is clearly not a subset of X (for two reasons; OutputStream is not a subset of InputStream, and loc#3 dead is not a subset of loc#3 live). The two states are then merged, and the result, stack={Stream}, loc#3=dead, is stored in X's EntryState. X is then re-visited. It should be noted that if X was inside one or more try blocks, then all filters and handlers for those blocks would also have to be queued for revisiting, since the initial state of X has changed.

[0139] There can often be a ripple effect when revisiting basic blocks; if instead of a return instruction, execution proceeded from X into another basic block, Y, it is quite likely that the current state would not be a subset of the state originally stored for Y either (since Y's original state was probably derived from X's original state), so Y would then be revisited.

[0140] Exceptions

[0141] When inside a try block, in theory, execution could jump to the exception handler/filter at any time. Therefore the verifier must determine the most conservative values of the local variables and argument slots through all points in the try block. The contents of the stack are not tracked, since the stack is cleared if an exception causes execution to enter the exception handler.

[0142] Primitive and value class locals and arguments (e.g. I4's, I8's, etc.) have a fixed type, and effectively only one bit of state—either "live" or "dead". Moreover, they can never revert from "live" to "dead" status. For example, once you have stored a "72" into an integer local variable, it is impossible to revert that local to containing uninitialized data. Therefore, the state of all primitive and value class locals at the start of the exception handler/filter will be the same as at the beginning of the try block. Argument slots do not need to be tracked, because arguments are always live, and their type is always fixed. For example, if the method signature states argument #3 is a Stream, it is acceptable to store an InputStream (subclass) into that argument, but as far as the verifier is concerned, whenever you read argument #3, you will get back a Stream, not what you stored into it.

[0143] Thus, the verifier needs to track only polymorphic locals—in other words, locals which are object references or byref object references. For example, if local variable #5 is declared as type Object in the locals signature, contains an InputStream at the start of the try block, but has an OutputStream stored into it at some point inside the try block, the verifier must "and" together these constraints to determine a conservative type for local variable #5 at the start of the exception handler. In this situation, the closest common superclass of InputStream and OutputStream would be found (for example, "Stream"). To accomplish this, the verifier traps the "stloc" function, since that is the only instruction which can change the type of a polymorphic local variable.

[0144] If a local was uninitialized at the beginning of the try block, then it will be considered uninitialized (dead) in

the handler. If verifying a value class constructor, the state of the field initialization status bitmap (indicating which fields of the value class have been initialized) at the beginning of the try block is propagated to the exception handler. It is illegal to have a try block around code where the "this" pointer is uninitialized, if "this" is an objref. The initialization status of "this" would be indeterminate inside the exception handler if this were allowed. In the value class case, the verifier will conservatively assume that "this" was uninitialized—it can do this, because it is allowable to construct a value class more than once, whereas it is illegal to construct an object reference more than once.

[0145] Byref

[0146] It is illegal to return a byref from a method if it was not a byref to something which had a permanent home (i.e. the Item in question had the "VER_FLAG_PERMANENT_HOME" flag set). Items which have permanent homes include static and instance fields of object references, and subfields of these fields (in the case where the container field is a value class). This relies upon the assumption that a byref to an object reference can prevent that object from being garbage collected. Everything else is considered to not have a permanent home. The addresses of local variables and stack slots are definitely not a permanent home.

[0147] Global/Local Types

[0148] All arguments and local variables have a global type, which is derived from the method signature and the locals signature respectively. It is illegal to store an incompatible type into a local or argument. For example, if a local is declared to be a String, it is illegal to store an InputStream into it. Generally, object locals will tend to be declared to be of type Object, to minimize the size of the signature, and to allow better signature pooling.

[0149] Arguments are not dynamically tracked. It is possible to store data into arguments, but reads from arguments always return the declared type. For example, if an argument is declared to be a Stream, it is possible to store an InputStream (subclass) into the argument. However, reading from the argument will always return a Stream.

[0150] Arguments therefore have only a global fixed type. Furthermore, they have no "current state" since they are always live. There is only one additional piece of information, which is whether the "this" pointer (argument slot #0) is initialized. The function Verifier::GetCurrentValueOfArgument() abstracts this away.

[0151] Local variables which contain primitive types or value classes also have only a global type. The only information stored for these is whether the local is live or dead (uninitialized), and this information is stored in a bitmap. Local variables which are not primitive types or value classes (e.g. object reference, or byrefs of primitive types or value classes) do NOT work this way, and their simulated type is fully tracked. These local variables therefore have a "current type" in addition to their "global type".

[0152] Each local variable is assigned a "slot". Primitive or value class locals are given negative slot numbers, -1, -2, -3, . . . , while all other locals are given positive slot numbers 0, 1, 2, These numbers are independent; one local can have a slot -1 and another local can have a slot of +1. The negative slot numbers correspond to bits in the "local

variable liveness bitmap"—here, a local will have a bit set if it is live, otherwise it is considered dead. Slot -1 corresponds to bit #0, slot -2 to bit #1, and so forth.

[0153] The macro `LIVEDEAD_TRACKING_ONLY_FOR_SLOT()` returns whether a given slot number indicates that only live/dead information is tracked for this local (all the macro does is return whether the slot number is negative). The macro `LIVEDEAD_NEGATIVE_SLOT_TO_BITNUM()` turns a negative slot number such as -3 into its corresponding bit number #2.

[0154] Locals which are stored in positive slot numbers have a full "Item" allocated for them, allowing complete type information to be stored (for example, whether the local is a byref of a local variable or instance field, is a pointer to something with a permanent home, etc.). This information is stored in the `m_pNonPrimitiveLocArgs` array, representing the current state of these locals. The function `Verifier::GetCurrentValueOfLocal()` abstracts away how the local information is stored, and simply returns an Item containing the current value of the local (which may be a "dead" type, if the local was dead).

[0155] There is also the concept of a "pinned type". If an object local variable has its address taken anywhere in the method (this is determined in the first pass over the code), its type is pinned to its globally declared type. There is a separate bitmap indicating which object locals have a pinned type. Pinned type is actually separate from the live/dead bitmap, even though it should probably not be. For EntryStates (states associated with basic blocks), information is stored in a similar array in the EntryState structure, as is a bitmap for all the locals for which only live/dead tracking is done.

[0156] Verifier Table

[0157] In order to simplify the code, the handling of a majority of instructions is handled via a table. For example, "add.i4" (a deprecated instruction, but a good example) has an operation string "44:4", meaning pop an I4, pop another I4, then push an I4. The typeless "add" instruction has the operation string "N:-", where "N" means "pop a number" (I4/I8/R4/R8/I), "=" means "pop another item of the stack, and it must be the same type as the previously popped item", and "-" means backtrack the stack by one. The ":" is a delimiter to separate pushing and popping operations.

[0158] CompatibleWith

[0159] The `Item::CompatibleWith` function is an important function in the verifier, and returns whether one item is compatible with (i.e. a subset of) another Item. The functionality of the item `CompatibleWith` is shown in FIG. 7, as can be appreciated by those of ordinary skill within the art.

[0160] Merge

[0161] The `Item::MergeToCommonParent` function is also an important function. However, it is not called very often—merging occurs only when the verifier needs to re-visit a basic block, because of a state mismatch, and that is a fairly rare situation. Merge takes two Items and finds the highest common factor. Unlike `CompatibleWith()`, merging is symmetric; A merged with B provides the same result as B merged with A. Examples are shown in FIG. 8, as can be appreciated by those of ordinary skill within the art.

[0162] Dirty Basic Block List

[0163] The dirty basic block list is a bitmap, with one bit for each basic block. All basic blocks which need to be visited have a bit set. Initially the entire bitmap is zeroed, and bits are set as control flow instructions are seen on the second pass. All basic blocks which have a bit set must have an EntryState associated with them, so that the verifier can simulate the machine state at that point.

[0164] Constructors

[0165] The rules for verifying constructor methods are different for value class constructors and object constructors.

[0166] For object constructors, it must be verified that the method calls either its direct superclass constructor, or an alternate constructor in the same class. An exception to the rule is the constructor for `Microsoft/Runtime/Object`, which has no superclass to call, and therefore the object is considered fully constructed upon entry into the constructor. The initialization state of the "this" pointer is part of the verifier state, and it checks that "this" cannot be passed to other methods until it has been constructed. It also ensures that when the "ret" instruction is encountered, "this" is initialized. It is illegal to construct an object more than once. Therefore if there is a path merge and one path constructed "this" and the other did not, verification fails.

[0167] For value class constructors it must be verified that all fields of the value class are initialized explicitly. This is because value classes can be stack allocated, and are therefore not zero-initialized (unlike objects allocated from the garbage collection heap). The current implementation is intentionally lenient about this, and enforces this rule only if there are any fields containing object references or value classes which themselves contain object references.

[0168] If a value class itself has value class fields, it is more tricky to verify its constructor, since such fields are initialized by loading the address of the field onto the stack (`ldflda`), possibly some parameters, potentially followed by control flow, and eventually calling a constructor on the field. The verifier must track that the address of the field is tied to an instance field, and implicitly consider the instance field to have been initialized by the call to the field's constructor.

[0169] Sloppy IL Code

[0170] It is common to see a local variable declared as an I4, performing a "ldloca" to load its address onto the stack, and then performing an operation as if it were a `Microsoft/Runtime/Boolean`. This is illegal, since an "&I4" is on the stack, not an "&I1". The solution for code generators is to properly declare the local as an "I1" or "`Microsoft/Runtime/Boolean`".

[0171] In a constructor for a value class, code sometimes initializes some of the fields, then calls another instance method to initialize other fields (e.g. "`this.count=GetCount()`"). In theory, "this" is not considered constructed until all fields have been written to, and therefore instance methods cannot be called on it until it is done. In practice, the verifier is intentionally lenient, and as long as there are no object references (including object references embedded in value class fields) in the value class, it allows it. Effectively, it is allowing garbage (uninitialized) values to exist in integer/float/etc. fields of value classes.

[0172] Specification of COM+ Run-Time Verification

[0173] In this section of the detailed description, a specification of a COM+ run-time verification that can be performed by a system or a method as previously described is provided. The description provided in this section is such that it is understandable to those of ordinary skill within the art. It is noted, however, that the invention itself is not limited to the details provided herein in this section.

[0174] Introduction to Verification

[0175] A file containing code for the COM+ Runtime may be transferred across a network to run in an Execution Engine hosted on a remote computer. The Execution Engine might be embedded in a web browser, for instance. The Execution Engine may or may not trust such mobile code to be safe to run. In the case that the mobile code is untrusted, the code is verified statically, that is, before it is allowed to run, to determine whether certain conditions hold. These conditions prevent execution errors such as the following:

[0176] type errors (such as confusing integers and pointers, or calling a method that does not exist);

[0177] access control errors (such as accessing a private field or method from outside its class);

[0178] object initialization errors (such as accessing a freshly created object before it has been appropriately initialized);

[0179] other errors such as stack overflows and references to local variables or arguments that do not exist.

[0180] Such execution errors could be exploited by untrusted mobile code to circumvent other security checks. In this way, untrusted mobile code could mount attacks on the host computer. The Execution Engine only runs untrusted code if it passes the checks performed during verification, which rule out these execution errors. Therefore, verification is an important mechanism for securing the COM+ Runtime when it runs untrusted code. Moreover, the verification checks may help to uncover code generation bugs in compilers that target the COM+ Runtime.

[0181] The verification task divides into two parts:

[0182] Metadata verification ensures that the classes and other items specified in a file make sense given the state of the Execution Engine which loads the file. Metadata verification is principally implemented by the class loader. It is performed when a file containing untrusted code is first loaded by the Execution Engine.

[0183] IL verification ensures type correctness of the IL instruction streams implementing each untrusted method. IL verification is implemented by the IL verifier. The Execution Engine invokes the IL verifier the first time it calls each untrusted method, and only executes the method if the IL verification succeeds.

[0184] The IL verifier is sometimes referred to simply as the verifier, which may be a source of confusion since its task is to perform just IL verification but not metadata verification. In fact, the verifier in one embodiment works on the assumption that metadata verification has been performed by other components of the Execution Engine. Both parts of the verification task in one embodiment are needed to avoid the execution errors mentioned above.

[0185] Metadata Verification

[0186] The purpose of metadata verification is to ensure that executable files received by the COM+ Runtime are well formed, that is, have the structure expected by the Runtime, and that the classes and other items they specify make sense in the environment into which they are loaded. This helps ensure that later processing of the items through IL verification, building method tables, interpretation, JITting (viz., just-in-time compiling), marshalling and so on, as known within the art, will not cause the COM+ Runtime to crash or behave in an insecure fashion. Many of the conditions checked by the verifier are simple structural conditions relating to the Execution Engine (EE) file format; however, some modestly complex checks are required to ensure the integrity of the data typing environment of the EE and to ensure the correctness of various execution and layout strategies. Thus, metadata verification refers to various structural conditions and wellformedness conditions, as is now described. Metadata itself thus refers to structure and wellformedness of the IL code.

[0187] Metadata: Structural Conditions

[0188] The COM+ Runtime should never crash when attempting to load and/or run an untrusted executable, no matter how badly the executable is formatted. The COM+ Runtime can make no assumptions that untrusted executables will be in the format specified in other COM+ Runtime documents, or that they will have been generated by tools like *assem* or a compiler for a high-level language. Therefore, the first set of checks to be performed simply ensure that the structural format of the data being read is valid. The COM+ loader must fail gracefully if these checks are not satisfied.

[0189] Metadata: Wellformedness Conditions for Metadata

[0190] The structural validity of a module is not sufficient to ensure that it is acceptable to load into the system. For example, modules that specify circular class hierarchies are not admissible, and untrusted modules must be checked for such properties. Only metadata that influences the behavior of the COM+ Runtime needs to be verified. Examples include the class and interface hierarchy, which determines what is a subtype of what.

[0191] Metadata Wellformedness: Resolution of Refs to Defs

[0192] Whenever type, member, or other "ref" (reference) tokens are encountered during either metadata or IL verification, they should successfully resolve to a unique "def" (definition). The "def" returned by resolution should be invariant after the first resolution of a "ref", e.g., further loading of classes should not alter the behavior of the resolution process. In principle, the COM+ Runtime might check the resolvability of refs and defs at any stage during the loading process, but in practice it happens "on-demand", i.e., as late in the loading process as possible.

[0193] Resolving a type reference might cause the class loader to be invoked recursively, which might, in turn, cause the metadata and IL verification algorithms to be invoked recursively. Mutually referential modules can be loaded and verified because the loading process performs a graph walk through the referential structure.

[0194] Metadata Wellformedness: Type Definitions

[0195] For each type definition, i.e., each `mdtTypeDef` metadata item, conditions such as the following are checked:

[0196] The name of the type definition is unique within its namespace.

[0197] The type is marked as an interface, value class, enumeration, or class. At present, this means at most one of `tdInterface`, `tdValueClass` and `tdEnum` can be set, although the exact encoding of this data might change.

[0198] If the type is a class, then:

[0199] All of its associated superclass, method, field, property, event, and “implemented interface” tokens resolve to metadata items of the correct kinds.

[0200] It extends some type, which must be a class. If the type is `Microsoft/Runtime/Object` it must not extend anything.

[0201] No circularity in the class hierarchy is induced.

[0202] The superclass is not final.

[0203] If any of the exposed methods of a class are abstract, then the class itself is abstract. The exposed methods are any public or protected methods defined in the class, or inherited from a superclass and not overridden by a method within the class.

[0204] It implements a set of types, each of which must be an interface. Furthermore, the exposed methods of the class must include the methods necessary to implement the interfaces, depending on the kind of the implementation relationship (`itImplements`, `itEvents`, `itRequires`, `itInherits`) for each interface.

[0205] If the type is an interface, then:

[0206] All of its associated method and interface tokens resolve to metadata items of the correct kinds.

[0207] It does not extend anything, but has an inheritance relation with zero or more types, each of which is an interface.

[0208] No circularity in the interface hierarchy is induced.

[0209] It is not final or abstract, has no non-static fields, and all its methods are public, virtual and abstract.

[0210] It contains no constructors

[0211] If the type is a value class, then:

[0212] See the Value class documentation for the list of constraints, e.g. the class must be derived directly from `Microsoft.Runtime.Value`, be immutable, be final, and so on.

[0213] If the type is an enumeration, then:

[0214] It only has static final values.

[0215] The type may have any kind of layout (automatic, sequential or explicit), however if explicit then the layout must not specify overlapping fields.

[0216] Metadata Wellformedness: Field Definitions in Types

[0217] For each field within a type, i.e., each associated `mdtFieldDef` metadata item, conditions such as the following are checked:

[0218] Taken together, the name and signature of the field must be unique within the type.

[0219] It must be one of private, protected, public or package, as known within the art.

[0220] It must have calling convention “field.”

[0221] Metadata Wellformedness: Method Definitions in Types

[0222] For each method within a type, i.e., each associated `mdtMethodDef` metadata item, conditions such as the following are checked:

[0223] The name/signature is unique within the type.

[0224] Its access rights must be one of private, protected, public or package.

[0225] It is static, virtual, abstract-virtual or none of these, and its calling convention not field.

[0226] If the method overrides a method in a superclass or interface (i.e., a non-private virtual method exists with the same name/signature as a new virtual method) then the overridden method is not final. From the perspective of EE type system, non-virtual methods are not considered to override virtual methods, and vice-versa: this is a source language choice.

[0227] The method has an implementation if and only if it is not abstract

[0228] If the method has an implementation, then that implementation is managed, non-optimized IL

[0229] The return type is not `RefAny`. It may be `ByRef(T)` but the IL verifier checks that only certain `ByRefs` are returned.

[0230] No argument or local has type `ByRef(ByRef(T))` or `ByRef(RefAny)`.

[0231] In addition, constructors must have return type void, must not be virtual or abstract.

[0232] Metadata Wellformedness: Events

[0233] For each event within a type, the following conditions are checked:

[0234] The name is unique within the type.

[0235] It has an Add method for the event name and that method takes a delegate type T and returns void.

[0236] It has a Remove method that takes the same delegate type T and returns void.

[0237] Optionally, it has a Raise method that takes an arbitrary set of parameters and returns void.

- [0238] The Add and Remove methods must have the same accessibility (public, protected, . . .). There is no restriction on the Raise method's accessibility.
- [0239] The delegate type T must have a void return type and be multicast. There are no other restrictions on the delegate's shape, including what parameters it takes and what exceptions it throws.
- [0240] Metadata Wellformedness: Global Methods
- [0241] For each global function, conditions such as the following are checked:
- [0242] The function must be neither abstract nor virtual and must be static.
- [0243] The function must have an implementation that satisfies the following requirements:
- [0244] It is managed, non-optimized IL
- [0245] IL Verification
- [0246] The purpose of IL verification is to guarantee that the IL code in an untrusted method is safe to execute, that is, that it will not give rise to certain unsafe execution errors including type errors, access control errors, and object initialization errors. In principle, the Execution Engine could avoid such execution errors by performing dynamic checks during the course of executing the untrusted method. However, this would degrade execution speed. Instead, the IL verifier according to one embodiment of the invention performs static checks to guarantee the absence of unsafe execution errors during execution of the untrusted method. These checks are static in the sense that they are performed before execution of the method begins. If IL verification succeeds, the verifier allows the method to execute. If IL verification fails, the method call fails, and the verifier raises the exception `Microsoft.Runtime.VerifierException`.
- [0247] The IL verifier makes two passes over the method:
- [0248] In the first pass, the verifier performs syntactic checks to ensure that the byte codes of the method can be parsed as a valid sequence of IL instructions. Every instruction in the method is visited during this pass.
- [0249] In the second pass, the verifier performs semantic checks to ensure that no execution path can lead to one of the execution errors described above. In this pass, the verifier performs a symbolic execution of the method, in which actual values are simulated by their types. Only instructions that are reachable by some execution path are visited during this pass.
- [0250] IL verification succeeds if and only if both the syntactic and semantic checks succeed.
- [0251] IL Verification: Types
- [0252] "Types" from the point of view of verification are not the same as the "datatypes" described in the COM+ Execution Engine Architecture description provided below. The typing scheme described in the COM+ EE Architecture deals only with basic types such as `I1`, `I`, `O`, & and so on: this typing scheme should always be obeyed, even by trusted code, since the correct operation of the (just-in-time compiler) JIT depends on it.
- [0253] Verification types are more specific than these. In particular `O` and & types include more information, that is, which class, array or interface is being referred to by a reference, and which type is being referred to by a `ByRef`. Further, `VC(td)` is used to indicate a type definition that is being used as an (unboxed) value class. Note that like the EE, the verifier draws no distinction between signed and unsigned types. The different types that appear at different places during verification are summarized in the table of FIG. 9(a) and FIG. 9(b). Note that the EE data type "*" does not arise during verification. Furthermore, the legend to be used in the table of FIG. (a) and FIG. 9(b) is as follows:
- [0254] M9: no, but possible
- [0255] X: Yes
- [0256] B: A "ret" instruction should return a result of type &*(ty) if the method is declared to return an &(ty); it may not return an &(ty); in the calling method, the value resulting from the call then has type &(ty) (and not &*(ty))
- [0257] C: What is done with values classes on the stack is to pass them to a method that is expecting a value class as a parameter, duplicate them or pop them off
- [0258] VC(I1): VC(Microsoft/Runtime/SignedByte)
- [0259] VC(U1): VC(Microsoft/Runtime/Byte), etc.
- [0260] VC(R4): VC(Microsoft/Runtime/Single)
- [0261] VC(R8): VC(Microsoft/Runtime/Double)
- [0262] Certain types arise during verification that are not allowed in method or field signatures. The main purpose of these is to track special situations concerning the initialization of locals and the correct behavior of constructors. These special situations are described next.
- [0263] IL Type Verification: Uninitialized Locals
- [0264] Uninitialized locals occur when the `ZeroInit` flag is false. Addresses to uninitialized value classes must be placed on the stack and appropriate constructors called, by using `ldarga. &Uninit(n,ty)` indicates a byref-pointer to a local variable containing a value class that is about to be initialized. Special types for verifying the correct use of uninitialized locals are shown in FIG. 10.
- [0265] IL Type Verification: Constructors Must Construct
- [0266] The special type `UnconThis` is used as the type of the uninitialized class instance that gets passed to non-value class constructor methods. In particular, argument 0 of such constructors is initially given this type. After the constructor calls its superclass constructor, or a constructor of the same class, argument 0 assumes the type `O(td)`, where `td` is the constructor's class. To ensure that the object has in fact been constructed, returning from a constructor is only allowed if argument 0 no longer has the type `UnconThis`. A summary of this type is shown in FIG. 11.
- [0267] IL Type Verification: Value Class Fields Must be Initialized
- [0268] For value class constructors, the construction may be delegated as before, but if not then every field of the value class object must be initialized before the constructor returns (regular objects have all their fields zero-initialized, but for value classes, which are allocated on the stack, there is no such guarantee). Thus, an initialization flag is kept for each field in the object. The object is considered constructed when all of these flags are set, at which point argument 0 assumes the type &(VC(td)), where `td` is the constructor's class. This is summarized in FIG. 12.

[0269] IL Type Verification: Only Special Byref's May be Returned

[0270] Byrefs that can be verified to be pointing into the heap, or to static data, or to any such "home" that will still exist after the current method exits (e.g. those passed in as arguments to the method), may be returned from a method. Byrefs which are known to have this property are marked &*(ty). This is summarized in FIG. 13.

[0271] IL Type Verification: Compatibility for Types

[0272] The compatibility relation for verification types (<:) is shown by example in FIG. 14. The diagram of FIG. 14 assumes class Bar implements interface BarI. Any types not shown do not exhibit any compatibility with other types.

[0273] IL Verification: Input to the Verifier

[0274] The Execution Engine provides the verifier with the following information about the method to be verified. (This information comes from the methoddef metadata item and the ILMethod metadata item for the method.) From the methoddef item comes the input:

[0275] The signature, including the number of fixed arguments, NumArgs

[0276] Whether the method is static, virtual, or neither.

[0277] Whether the method is public, protected, or private.

[0278] Whether the method is a constructor.

[0279] Whether the method is a class constructor.

[0280] Furthermore, from the ILMethod item comes the input:

[0281] The maximum height of the stack, referred to as MaxStack.

[0282] The tailsig that specifies the number (NumLocals) and types of local variables.

[0283] An array of bytes, referred to as codes, and its length.

[0284] An array of exception clauses, where an exception clause specifies Start, End, Handler and Finally offsets and either a Filter offset or a ClassToken.

[0285] The ZeroInit flag for the method

[0286] The verifier also depends on metadata information from the COM+ Runtime environment. In particular, it depends on the method and field signatures of all members accessed from the method, as can be appreciated by those of ordinary skill within the art.

[0287] IL Verification First Pass (Syntactic Checks)

[0288] The syntactic checks determine whether the codes array consists of a well formed sequence of IL instructions. The syntactic checks apply to every IL instruction parsed from the codes array, whether or not there is an execution path that reaches it. The syntactic checks are as follows:

[0289] The length of the codes array is greater than 0.

[0290] The codes array can be parsed into an array of IL instructions according to the rules in the "IL Opcode Encoding" specification, such that the last byte of the last parsed IL instruction is the last byte of the codes array

[0291] The target of each branch or leave instruction lies within the codes array at the start of a parsed IL instruction. Each target of each switch instruction lies within the codes array, and at the start of a parsed IL instruction.

[0292] For every tailcall instruction, the following instruction must be a call or callvirt and there must be no branches directly to the following instruction, and no exception handler can specify the following instruction as a target (i.e. tailcall must be a true annotation, and it must be possible to treat the tailcall/call/callvirt instructions as an atomic instruction).

[0293] IL Verification Second Pass (Semantic Checks)

[0294] The IL verifier performs the semantic checks by simulating the effect of executing the instructions in the method. The types of values that are tracked may be assumed by the arguments, stack slots, and local variables over the course of any execution path through the method, and check that no instruction can give rise to a type error. The definitions shown in FIG. 15 are used to specify the semantic checks.

[0295] IL Verification Second Pass (Semantic Checks): Type States

[0296] A type state records constraints on the types of arguments, locals and stack slots.

[0297] A type state contains:

[0298] args. An array of types, one for each argument to the method. (The type of each argument is fixed throughout the method, except that in a constructor, argument 0 starts as the type UnconThis, but after initialization it becomes either &(VC(this)) for a value class this constructor, or O(this) for a class this that is not a value class.)

[0299] locals. An array of types, one of each for each local variable. (The type of local i may vary, unless there is a ldloc i anywhere in the method.)

[0300] stack. An array of types indicating the current state of the argument stack.

[0301] fields. Flags used to indicate which fields a value class constructor has initialized.

[0302] block. Flags recording if were in a regular, finally, filter or handler block.

[0303] tailcall-ok. Flag recording whether it is safe to perform a tailcall. It is safe to perform a tailcall if no addresses of locals or arguments have been taken.

[0304] A summary of operations on type states is shown in the table of FIG. 16.

[0305] IL Verification Second Pass (Semantic Checks): The Basic Verification Algorithm

[0306] In one embodiment, the semantic checks can be expressed as an algorithm, which acts on the following state.

[0307] The Pending Nodes in the Dataflow Analysis: a set of pairs (i, state), where i is an address, and state is a type state. This set stores those locations still to be explored in the data flow analysis of the method. Note that locations may be revisited with refined type states.

- [0308] The Visited Nodes in the Dataflow Analysis: as addresses are analyzed, the results of the analysis are recorded, and thus each address is either unmarked, or marked with a type state.
- [0309] In one embodiment, the algorithm is as follows:
- [0310] 1. Initialize Pending to be the singleton set $\{(0, \text{state}0)\}$ where $\text{state}0$ has $\text{args} =$
- [0311] $[\text{arg}1', \dots, \text{arg}n']$ if the method is static, with
- [0312] $[\text{UnconThis}, \text{arg}1', \dots, \text{arg}n']$ if the method is a constructor.
- [0313] $[\text{O}(\text{id}), \text{arg}1', \dots, \text{arg}n']$ otherwise
- [0314] Here the signature of the method is $\text{class}::\text{method}(\text{arg}1, \dots, \text{arg}n)$ with $\text{arg}i'$ being $\text{arg}i$ with $\&(\text{ty})$ types replaced by $\&*(\text{ty})$. $\text{locals} =$
- [0315] $[\text{loc}1, \dots, \text{loc}n]$ if the ZeroInit flag is true.
- [0316] $[\text{Uninit}(\text{loc}1), \dots, \text{Uninit}(\text{loc}n)]$ if the Zero-Init flag is false.
- [0317] Here the tailsig of the method is $[\text{loc}1, \dots, \text{loc}n]$ $\text{stack} = \square$ $\text{fields} =$
- [0318] $[\text{false}, \dots, \text{false}]$ if verifying a value class constructor, one for each field.
- [0319] undefined otherwise $\text{block} = \text{regular}$ $\text{tailcall} = \text{ok} = \text{true}$
- [0320] 2. If Pending is the empty set, halt with success.
- [0321] 3. Choose a pair (i, state) from the set Pending, and remove it from the set.
- [0322] 4. If the address i is marked with the state old :
- [0323] If the least upper bound $\text{old} \cup \text{state}$ exists, then:
- [0324] If $\text{old} \cup \text{state}$ equals old , goto 2.
- [0325] Otherwise, unmark the address i , insert $(i, \text{old} \cup \text{state})$ into Pending, and goto 2.
- [0326] Otherwise, halt with failure.
- [0327] 5. If the address i is unmarked:
- [0328] Mark the address i with the state state .
- [0329] Let inst be the IL instruction whose first byte is at address i .
- [0330] The effect an instruction has on the type state as has been defined. If any of the checks described there fail, then halt with failure. Otherwise, let state' be the outcome of running inst on state .
- [0331] If the length of the stack component of state' is greater than MaxStack , halt with failure.
- [0332] Insert one element (j, state') into Pending for each destination address j specified in the detailed description below. In addition, for each exception clause with Start , End , Handler , and Finally offsets, where $\text{Start} \leq i < \text{End}$, and a class token Class , we insert $(\text{Handler}, \text{state}')$ and $(\text{Finally}, \text{state}')$ into Pending, where state' is state' with the stack set to $[\text{Class}]$. If the exception clause has a Filter offset, then insert $(\text{Filter}, \text{state}')$ where state' is state' with the stack set to $[\text{Microsoft/Runtime/Object}]$
- [0333] Goto 2.
- [0334] The algorithm terminates because the number of elements in the state never exceeds the total number of instructions in the method, and on each iteration the type state for at least one instruction is increased according to the $<$: ordering, and this ordering is finite.
- [0335] IL Verification Second Pass (Semantic Checks): Effect of an Instruction on Control Flow
- [0336] The destination addresses of an instruction are as specified in the table of FIG. 17, where next is the subsequent instruction. For instructions not appearing in the table, the destination address is simply next. Here k indicates the address referred to by the offset in the instruction.
- [0337] IL Verification Second Pass (Semantic Checks): Effect of an Instruction on Type State
- [0338] In this section, the effect of each instruction has on the type state is described. Each instruction is regarded as transforming a given type state into another, which is referred to as the outcome of running the instruction. Almost all of the instructions only affect the stack, and have no effect on the arguments and locals. Unless otherwise stated, the stack is the only component of the state that changes.
- [0339] If the stack typing for an instruction is defined in FIGS. 18, 19(a), 19(b) and 20 as $[\text{ty}1, \dots, \text{ty}n] \Rightarrow \text{pushes}$, then the effect of an instruction is only defined if stack has $\text{shape} \text{ stack} = \text{base} + [\text{pop}1, \dots, \text{pop}n]$, and $\text{pop}i < \text{ty}i$ for $1 \leq i \leq n$. That is, $[\text{ty}1, \dots, \text{ty}n]$ specifies a series of constraints that must be satisfied up to subtyping. The new stack is $\text{base} + \text{pushes}$. That is, pushes specifies exact types rather than types up to subtyping.
- [0340] The effect of various control flow instructions on the type state is shown in the table of FIG. 18. The effect of various basic instructions on the type state is shown in the table of FIG. 19(a) and FIG. 19(b). The effect of various object model instructions, excluding methods and fields, is shown in the table of FIG. 20.
- [0341] For method call instructions, each of the following cases assumes md resolves to a method with signature $\text{rty } T::\text{method}(x1, \dots, xn)$ where rty is the return type. In each case, if md has protection protected then it must be $\text{this} <: T$, where this is the type containing the method being verified. Similarly, if md has protection private then $\text{this} = T$. Thus, the effect of various method call instructions, with these assumptions in mind, is shown in the table of FIG. 21(a) and 21(b).
- [0342] For field instructions, each of the following cases assume fd resolves to a field with signature $\text{fty } T::\text{field}$. In each case, if fd has protection protected then it must be $\text{this} <: T$, where this is the type containing the method being verified. Similarly, if fd has protection private then it must be $\text{this} = T$. Thus, the effect of various field instructions, with these assumptions in mind, is shown in the table of FIG. 22.

[0343] Generating Verifiable Code: Verifiability and Optimizations

[0344] As can be appreciated by those of ordinary skill within the art, verifiability substantially limits the kind of code optimizations that may be used at compile time. This is because the responsibility for ensuring type soundness is shifted from the compiler to the verifier/JIT. Only optimizations where type soundness remains manifest in the code produced can be used when generating verifiable code. Optimizations such as loop unrolling are acceptable, because the code produced has essentially the same type structure as the original.

[0345] For example, if a compiler inlines across protection boundaries, then code may no longer be verifiable. Consider

```
[0346] class A {
    [0347] public: void add( ) {x=x+1;}
    [0348] private: int x;
[0349] }
[0350] class B {int main( . . . ) { . . . A a; a.add( ); . . . }
```

[0351] Many compilers would inline the `a.add()` call. However, to produce verifiable code, the compiler should not do this. This is because moving `x=x+1` into class B would mean a method in B was directly accessing a private field of A.

[0352] Similarly, optimizations such as the aggressive reuse of stack, local and variable slots should either not be used, or in the case of Opt-IL annotations, will be rejected or ignored for untrusted code.

[0353] Generating Verifiable Code: Verifiability and COM+ IL Features

[0354] This section gives a description of the features in the verifiable subset of IL and explains how they should be used in order to ensure the code produced by a compiler is verifiable. The basic rules for generating verifiable code are as follows:

[0355] The metadata, i.e., classes, interfaces, arrays, value classes, and other items defined in a module must satisfy the constraints of the Virtual Object System, i.e., they must make sense according to the rules of this system. The rules are fairly obvious, and the system will simply refuse to load classes that violate them (though these checks may not be fully implemented in the alpha), so most code generators will not be bothered by this.

[0356] Stack slots have types, and data manipulations involving the stack must respect these types. A simple inference process can check and reconstruct the basic JIT/EE datatypes of stack slots. Verification goes beyond this and verifies the correct use of Object reference types.

[0357] Argument slots have similar behavior to stack slots. The types of these slots may not change: they are always constrained to the type specified in the method signature. You may store a compatible type into the slot, but when you retrieve it the item on the stack will be considered to have the declared type.

[0358] Local slots are similar to argument slots, however the types of these slots may change, though their IL datatypes may not (verification types are more specific than IL datatypes). That is, a local initially holding a reference to a String can be later used to hold a reference to some other kind of object, but a slot used to hold an I cannot later hold an IS.

[0359] Initialization of locals. Argument slots are initially "live", local slots are "uninitialized" (unless the `ZeroInit` flag for a method is set). Uninitialized slots must be initialized, either by performing appropriate `stloc`'s or calling appropriate value-class constructor methods.

[0360] ValueClasses are relatively simple to verify: just remember that they act like base types. Value Class Constructors must initialize every field of the value class—the fields are initially uninitialized. Uninitialized fields cannot be used for any purpose except to take their address for initialization or to store into.

[0361] Branches must be to instructions within the same method. Moreover the types of the slots (locals, arguments, stack) must "match up" between the source and the destination. This means there must exist types such that (a) the assumptions encoded by those types are strong enough to ensure the code at the destination can be verified, and (b) the assumptions are weak enough that all control-transfers to that location guarantee they will be met.

[0362] Byref and Refany values are designed as a safe way to do some pointer access without using the object model. Byref and Refany types can only be used in a highly stylized way. Although they can point either into the heap or the stack, Byref values themselves always reside on the stack, and verification ensures we always know exactly where they live (garbage collection also needs to know this). They may only be returned from functions if the verifier can determine that the byref points into the heap. The most immediate use for compilers is to implement call-by-reference semantics for languages such as VB and Modula-2, and also to perform manipulations on value classes. Byrefs mean that many instructions that may appear "dangerous" are in fact verifiable when they use or produce arguments of this kind. In particular instructions that load addresses (`ldloca`, `ldarga`, `ldflda`, `ldsflda` etc.) are all verifiable, as are `ldind` and `stind`.

[0363] Taking the Address of Locals. Byrefs allow code to take the address of locals using `ldloca`. This includes locations whose type may, in theory, change, by a later assignment, e.g. object references. The verifier disallows this. If a `ldloca` is ever done on a local slot, anywhere within a method, then the type of that slot is constrained to its declared type (in the local signature) precisely as if it were an argument slot.

[0364] Access control (public, private, etc.) must be respected according to the rules of the VOS.

[0365] Constructors must not use the "This" pointer until they either call another constructor in their class or call a constructor of their superclass. Constructors must call such a constructor at some point before returning.

[0366] Exceptions require special care. The exception handling tables for a method specifies a series of possible control transfers, and each of these must satisfy the conditions for branching specified above.

[0367] COM+ Execution Engine Architecture

[0368] In this section of the detailed description, the architecture of a COM+ execution engine, which can be a part of a system as previously described, is presented. In particular, the architecture is described insofar and such that the verification aspects of embodiments of the invention already described are understandable to those of ordinary skill within the art. Thus, the description provided in this section is such that it is understandable to those of ordinary skill within the art. It is noted, however, that the invention itself is not limited to the details provided herein in this section.

[0369] Execution Engine Overview

[0370] The Execution Engine (EE) that manages the execution of source code compiled into Intermediate Language (IL), OptIL, or native machine code. All code based on COM+ IL executes as managed code, that is code that runs under a "contract of cooperation". The environment in which the EE operates is referred to as the COM+ Runtime environment, or simply COM+ Runtime.

[0371] A feature of the Execution Engine is its ability to provide software isolation of programs running within a single address space. It does this by enforcing typesafe access to all areas of memory when running typesafe managed code. Some compilers generate IL that is not only typesafe but whose typesafety can be proven by simply examining the IL. This process, verification, allows servers to quickly examine user programs written in IL and only run those that it can demonstrate will not make unsafe memory references. This independent verification is critical to truly scalable servers that execute user-defined programs (scripts).

[0372] The EE provides the following services:

- [0373] Code management
- [0374] Software memory isolation
- [0375] Verification of the typesafety of IL
- [0376] Conversion of IL to native code
- [0377] Loading and execution of managed code (IL or native)
- [0378] Accessing metadata (enhanced type information)
- [0379] Managing memory for managed objects
- [0380] Insertion and execution of security checks
- [0381] Handling exceptions, including cross-language exceptions
- [0382] Interoperation between COM+ objects and legacy COM objects
- [0383] Automation of object layout for late binding
- [0384] Supporting developer services (profiling, debugging, etc.)

[0385] The EE supplies the common infrastructure that allows tools and programming languages to benefit from cross-language integration.

[0386] One function of the EE is on-the-fly conversion of IL (or OptIL) to native code. Source code compilers generate IL (or OptIL), and JIT compilers convert the IL to native code for specific machine architectures. As long as a simple set of rules are followed by the IL generator, the same IL code will run on any architecture that supports the COM+ Runtime. Because the conversion from IL to native code occurs on the target machine, the native code can take advantage of hardware-specific optimizations. Other significant EE functions include class loading, verification, and support for security checks.

[0387] EE Overview: IL and OptIL

[0388] IL is a stack-based set of instructions designed to be easily generated from source code by compilers and other tools. Several kinds of instructions are provided, including instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and method invocation. There is also a set of IL instructions for implementing object-oriented programming constructs such as virtual method calls, field access, array access, and object allocation and initialization.

[0389] The IL instruction set can be directly interpreted by tracking the data types on the stack and emulating the IL instructions. It can also be converted efficiently into native code. The design of IL allows this process to produce optimized native code at reasonable cost. The design of IL allows programs that are not typesafe to be expressed, since this is essential for support of some common programming languages. At the same time, by following a simple set of rules, it is possible to generate IL programs that are not only typesafe but can easily be proven to be so.

[0390] EE Overview: Class Loading

[0391] The EE's class loader loads the implementation of a class into memory, checks that it is consistent with assumption made about it by other previously loaded classes, and prepares it for execution. To accomplish this task, the class loader ensures that certain information is known, including the amount and the shape of the space that instances of the type require. In addition, the class loader determines whether references made by the loaded type are available at runtime and whether references to the loaded type are consistent.

[0392] The class loader checks for certain consistency requirements that are vital to the COM+ security enforcement mechanism. These checks constitute a minimal, mandatory, verification process that precedes the IL verification, which is more rigorous (and optional). In addition, the class loader supports security enforcement by providing some of the credentials required for validating code identity.

[0393] EE Overview: Verification

[0394] Typesafe programs reference only memory that has been allocated for their use, and they access objects only through their public interfaces. These two restrictions allow objects to safely share a single address space, and they guarantee that security checks provided by the objects' interfaces are not circumvented. Code access security, the COM+ Runtime's security mechanism, can effectively protect code from unauthorized access only if there is a way to verify that the code is typesafe.

[0395] To meet this need, the COM+ Runtime provides a verifier, which uses the information in type signatures to help determine whether IL code is typesafe. It checks to see that metadata is well-formed, and it performs control flow analyses to ensure that certain syntactic and semantic conditions are met. The verifier declares that a program is successfully verified only if it is typesafe.

[0396] Used in conjunction with the strong typing of metadata and IL, verification can ensure the typesafety of programs written in IL. The COM+ Runtime requires code to be verified before it is run, unless a specific (administratively controlled) security check determines that the code can be fully trusted.

[0397] EE Overview: Security Checks

[0398] The EE is involved in many aspects of the COM+ Runtime's security mechanism. In addition to the verification process required by code access security, the EE provides support that enables both declarative and imperative security checks to occur.

[0399] Declarative security checks take place automatically whenever a method is called. The permissions that are required in order to access the method are stored in the component's metadata. At run time, calls to methods that are marked as requiring specific permissions are intercepted to determine whether callers have the required permissions. A stack walk is sometimes necessary to determine whether each caller in the call chain also has the required permissions.

[0400] Imperative security checks occur when special security functions are invoked from within the code being protected. The EE supports this type of security check by providing trusted methods that enable code identity to be determined and allow permissions to be located and stored in the stack. In addition, the EE gives the security engine access to administrative information about security requirements.

[0401] Virtual Execution System

[0402] The Execution Engine creates an environment for code execution called the Virtual Execution System, which is shown in FIG. 23. In most cases, source code is compiled into IL, and the IL is loaded and executed on-the-fly using one of the JIT compilers to convert the IL to native code. Note that for trusted code, verification can be omitted.

[0403] The EE's metadata engine enables the source code compiler to place metadata in the PE file along with the generated IL or OptIL. During loading and execution, this metadata provides information needed for registration, debugging, memory management, and security. Also indicated in the diagram is the fact that classes from the COM+ Base Class Library can be loaded by the class loader along with IL, OptIL, or native code.

[0404] Another execution path that can be chosen involves precompilation to native code using a backend compiler. This option might be chosen if compiling code at run-time is unacceptable due to performance requirements. As indicated in the diagram, precompiled native code bypasses verification and JIT compilation. Because precompiled native code is not verified, it must be considered fully trusted code in order to execute.

[0405] Supported Data Types

[0406] The Execution Engine directly supports the data types shown in the table of FIG. 24. These data types can be implemented using the IL instruction set. However, only the data types I4, I8, I, and R are supported by all numeric instructions. Instead, the support for the short values (I1, I2, R4, U1, and U2) is largely handled by conversion to wider values. This reflects a computer model that assumes static locations are 1, 2, 4, or 8 bytes wide but registers and stack locations are either 4 or 8 bytes wide. The support for short values consists of:

[0407] Load and store instructions to/from memory: ldelem, ldind, stind, stelem

[0408] Arithmetic with overflow detection: add.ovf, mul.ovf, sub.ovf

[0409] Data conversion: conv, conv.ovf

[0410] Loading constants: ldc

[0411] Array creation: newarr

[0412] The signed integer (I1, I2, I4, I8, and I) and unsigned integer (U1, U2, U4, U8, and U) types differ only in the how the bits of the integer are interpreted. For those operations where an unsigned integer is treated differently from a signed integer (e.g. comparisons or arithmetic with overflow) there are separate instructions for treating an integer as unsigned (e.g. cgt.un and add.ovf.u).

[0413] This instruction set design simplifies JIT compilers and interpreters of IL by allowing them to internally track a smaller number of data types.

[0414] Supported Data Types: Natural Size: I, R, U, O and &

[0415] The natural size types (I, R, U, O, and &) are a mechanism in the EE for deferring the choice of a value's size. These data types have a fixed but unknown size when the IL is generated at compile time. Instead, the choice of size is deferred until JIT compilation, when the EE has been initialized and the architecture is known. This implies that field and stack frame offsets are also not known at compile time. For languages like Java and Visual Basic, where field offsets are not computed early anyway, this is not a hardship. In languages like C or C++, a conservative assumption that they occupy 8 bytes is sometimes acceptable.

[0416] Supported Data Types: Unmanaged Pointers as Type U

[0417] For languages like C, where the size of a pointer is known at compile time and there are no managed objects, the fixed-size unsigned integer types (U4 or U8) can serve as pointers. However making the choice of pointer size at compile time has its disadvantages. If pointers were chosen to be 32 bit quantities at compile time, the code would be restricted to 4gig of address space, even if it were run on a 64 bit machine. Moreover, a 64 bit EE would need to take special care so those pointers passed back to 32-bit code could always fit in 32 bits. If pointers were chosen at compile time to be 64 bits, the code could be run on a 32 bit machine, but pointers in every data structure would be twice as large as necessary on that EE.

[0418] It is desirable, especially when building reusable library routines, to defer the choice of pointer size from compile time to EE initialization time. In that way, the same compiled code can handle large address spaces for those applications that need them, while also being able to reap the size benefit of 32 bit pointers for those applications that do not need a large address space.

[0419] For these reasons, the U type should be used to represent unmanaged pointers.

[0420] Supported Data Types: Managed Pointer Types: O and &

[0421] The O datatype represents an object reference that is managed by the COM+ runtime. As such, the number of legal operations is severely limited. In particular, references can only be used on operations that indicate that they operate on reference types (e.g. `ceq` and `ldind.ref`), or on operations whose metadata indicates that references are allowed (e.g. `call`, `ldsfld`, and `stsfld`).

[0422] The & datatype (managed pointer) is similar to the O type, but allows pointers to the interior of an object. That is, a managed pointer is allowed to contain the address of a particular field within an object rather than to the whole object. Managed pointers must be reported to the COM+ runtime memory manager (as must object references).

[0423] Supported Data Types: Portability: Storing Pointers in Memory

[0424] Several instructions, including `calli`, `cpblk`, `initblk`, `ldind.*`, and `stind.*`, expect an address on the top of the stack. If this address is derived from a pointer stored in memory, there is an important portability consideration.

[0425] 1. Code that stores pointers in a natural sized integer or pointer location (types I, O, U, or &) is always fully portable.

[0426] 2. Code that stores pointers in an 8 byte integer (type I8 or U8) can be portable. But this requires that a `conv.ovf.u` instruction be used to convert the pointer from its memory format before its use as a pointer. This may cause a runtime exception if run on a 32-bit machine.

[0427] 3. Code that uses any smaller integer type to store a pointer in memory (I1, U1, I2, U2, I4, U4) is never portable, even though the use of a U4 or I4 will work correctly on a 32-bit machine.

[0428] Supported Data Types: Handling of Floating Point Data Types

[0429] The Execution Engine assumes floating-point calculations are handled as described in the IEEE 754 standard, "IEEE Standard for Binary Floating-point Arithmetic," known within the art. This standard describes encoding of floating point numbers, definitions of the basic operations and conversion, rounding control, and exception handling.

[0430] The standard defines three special values, NaN, (not a number), +infinity, and -infinity. These values are returned on overflow conditions. A general principle is that operations that have a value in the limit return an appropriate infinity while those that have no limiting value return NaN,

but see the standard for details. For purposes of comparison, infinite values act like a number of the correct sign but with a very large magnitude when compared with finite values. NaN is 'unordered' for comparisons (see `clt`, `clt.un`).

[0431] While the IEEE 754 specification also allows for exceptions to be thrown under unusual conditions (overflow, invalid operand, . . .), the EE does not generate these exceptions. Instead, the EE uses the NaN return values and provides the instruction `ckfinite` to allow users to generate an exception if a result is NaN, +infinity, or -infinity. The rounding mode defined in IEEE 754 is set by the EE to round to the nearest number.

[0432] Supported Data Types: IL Instructions and Numeric Types

[0433] Most IL instructions that deal with numbers take their inputs from the evaluation stack, and these inputs have an associated type that is known to the JIT compiler. As a result, a single operation like `add` can have inputs of any numeric data type, although not all instructions can deal with all combinations of operand types. Binary operations other than addition and subtraction require that both operands must be of the same type. Addition and subtraction allow an integer to be added to or subtracted from a managed pointer (types & and *).

[0434] Instructions fall into the following categories: numeric, integer, floating point, specific, unsigned/unordered, and load constant instructions. Numeric instructions deal with both integers and floating point numbers, do not treat short integers in any special way, and consider integers to be signed. Simple arithmetic, conditional branch, and comparison instructions fit in this category. Integer instructions deal only with integers, but do not treat short integers in any special way. Bit operations and unsigned integer division/remainder fit in this category. Floating point instructions deal only with floating point numbers. Specific instructions deal with integer and/or floating point numbers, but have variants that deal specially with different sizes and unsigned integers. Integer operations with overflow detection, data conversion instructions, and operations that transfer data between the evaluation stack and other parts of memory fit into this category. Unsigned/unordered instructions are special comparison and branch instructions that treat integers as unsigned and consider unordered floating point numbers specially (as in "branch if greater than or unordered"). The load constant (`ldc.*`) instructions can be used to load constants of type I4, I8, or R.. Natural size constants (type I) must be created by conversion from I4 (conversion from I8 would not be portable) using `conv.i` or `conv.u`.

[0435] The table of FIG. 25 shows the IL instructions that deal with numeric values, along with the category to which they belong. Instructions that end in ".*" indicate all variants of the instruction (based on size of data and whether the data is treated as signed or unsigned).

[0436] Supported Data Types: IL Instructions and Pointer Types

[0437] The Execution Engine has the ability to track pointers to objects and collect objects that are no longer reachable (memory management by "garbage collection"). This process copies objects to compact the working set and thus must modify all pointers to objects as they move. For

this to work correctly, pointers to objects must only be used in certain ways. The O (object reference) and & (managed pointer) datatypes are the formalization of these restrictions.

[0438] The use of object references is tightly restricted in the IL. A few of the base instructions of the IL can handle object references. In particular, they can be:

[0439] Loaded/stored onto the stack to be passed as arguments to methods (ldloc, ldarg, stloc, starg)

[0440] Duplicated or popped off the stack (dup, pop)

[0441] Tested for equality with one another, but not other data types (beq, beq.s, bne, bne.s, ceq)

[0442] Loaded/stored into unmanaged memory, in type unsafe code only (ldind.ref, stind.ref)

[0443] Create a null reference (ldnull)

[0444] Returned as a value (ret)

[0445] Managed pointers have several additional base operations.

[0446] Addition and subtraction of integers, in units of bytes, returning a managed pointer (add, add.ovf.u, sub, sub.ovf.u)

[0447] Subtraction of two managed pointers, returning the number of bytes between them (sub, sub.ovf.u)

[0448] Unsigned comparison and conditional branches based on two managed pointers (bge.un, bge.un.s, bgt.un, bgt.un.s, ble.un, ble.un.s, blt.un, blt.un.s, cgt.un, clt.un)

[0449] Supported Data Types: Aggregate Data

[0450] The EE supports aggregate data, that is, data items that have sub-components (arrays, structures, or instances) but are passed by copying the value. The sub-components can include references to managed memory. Aggregate data is representing using a value class, which can be instantiated in two different ways:

[0451] Boxed: as an Object, carrying full type information at runtime, and typically allocated on the heap by the COM+ runtime memory manager.

[0452] Unboxed: as a "value class instance" which does not carry type information at runtime and which is never allocated directly on the heap. It can be part of a larger structure on the heap—a field of a boxed class instance or an element of an array. Or it can be on the stack as an argument or local variable. Or it can be allocated as a static variable or static member of a class.

[0453] Because value class instances are copied on method call, they do not have "identity" in the sense that Objects (boxed instances of classes) have; see the VOS specification.

[0454] Supported Data Types: Aggregate Data: Homes for Values

[0455] The home of a data value is where it is stored for possible reuse. The EE directly supports the following home locations:

[0456] 1. An incoming argument

[0457] 2. A local variable of a method

[0458] 3. A field of an object instance

[0459] 4. A static field associated with a class, interface, or module

[0460] 5. An array element

[0461] For each home location, there is a manner to compute (at runtime) the address of the home location and a manner to determine (at JIT compile time) the type of a home location. These are summarized in the table of FIG. 26. In addition to homes, built-in values can exist in two additional ways (i.e. without homes):

[0462] 1. as constant values (typically embedded in the IL instruction stream using ldc.* instructions)

[0463] 2. as an intermediate value on the evaluation stack, when returned by a method or IL instruction.

[0464] Supported Data Types: Aggregate Data: Opaque Classes

[0465] Some languages provide multi-byte data structures whose contents are manipulated directly by address arithmetic and indirection operations. To support this feature, COM+ allows value classes to be created with a specified size but no information about their data members. Instances of these "opaque classes" are handled in precisely the same way as instances of any other class, but the ldld and stld instructions cannot be used to access their contents.

[0466] Executable Image Information

[0467] The execution engine accesses metadata that is stored in a PE file, which is a portable executable file such as a .dll file or a .exe file, as known within the art. Specifically, the EE relies on the following information about each method defined in a PE file:

[0468] The instructions composing the method body, including all exception handlers.

[0469] The signature of the method, which specifies the return type and the number, order, parameter passing convention, and primitive data type of each of the arguments. It also specifies the native calling convention (this does not affect the IL virtual calling convention, just the native code).

[0470] The exception handling array. This array holds information delineating the ranges over which exceptions are filtered and caught.

[0471] The maximum operation stack (also called the evaluation stack) depth.

[0472] A "zero init flag" that indicates whether the local variables and memory pool should be initialized by the EE (see also localloc).

[0473] Type of each local variable in the form of a signature of the local variable array (called the "locals signature").

[0474] In addition, the file format is capable of indicating the degree of portability of the file.

[0475] There are two kinds of restrictions that can be described:

[0476] Restriction to a specific (32-bit or 64-bit) natural size for integers.

[0477] Restriction to a specific "endian-ness" (i.e. whether bytes are stored left-to-right or right-to-left within a machine word).

[0478] By stating what restrictions are placed on executing the code, the EE class loader can prevent non-portable code from running on an architecture that it cannot support.

[0479] Control Flow

[0480] The IL instruction set provides a set of instructions to alter the normal flow of control from one IL instruction to the next.

[0481] Conditional and Unconditional Branch instructions for use within a method, provided the transfer doesn't cross a protected region boundary.

[0482] Method call instructions to compute new arguments, transfer them and control to a known or computed destination method.

[0483] Tail call prefix to indicate that a method should relinquish its stack frame before executing a method call.

[0484] Return from a method, returning a value if necessary.

[0485] Method jump instructions to transfer the current method's arguments to a known or computed destination method.

[0486] Exception-related instructions. These include instructions to initiate an exception, transfer control out of a protected region, and end a filter, catch clause, or finally clause.

[0487] While the EE supports arbitrary control transfers within a method, there are several restrictions that should be observed, and which are tested by the verifier:

[0488] Control transfer is never permitted to enter a catch handler or finally clause except through the exception handling mechanism.

[0489] Control transfer out of a protected region is only permitted through an exception instruction (leave, end-filter, end.catch, or end.finally).

[0490] The evaluation stack must be empty after the return value is popped by a ret instruction.

[0491] All slots on the stack must have the same data type at every point within the method body, regardless of the control flow that allows execution to arrive there.

[0492] Control is not permitted to simply "fall through" the end of a method. All paths must terminate with one of these instructions: ret, throw, jmp, jmp, or tailcall followed by call, calli, or callvirt.

[0493] Method Calls

[0494] An important design goal of the EE is to abstract method frame layout, including calling convention. That is, instructions emitted by the IL code generator contain sufficient information for different implementations of the EE to use different native calling convention.

[0495] Method Calls: Call Site Descriptors

[0496] To support this flexibility, call sites need additional information that enables an interpreter or JIT compiler to synthesize any native calling convention. All IL calling instructions (call, calli, and callvirt) include as part of the instruction a description of the call site. This description can

take one of two forms. The simpler form, used with the calli instruction, is a "call site description" that provides:

[0497] The number of arguments being passed.

[0498] The primitive data type of each argument.

[0499] The order in which they have been placed on the call stack.

[0500] The native calling convention to be used

[0501] The more complicated form, used for the call and callvirt instructions, is a "method reference" that augments the call site description with an identifier for the target of the call instruction.

[0502] Method Calls: Calling Instructions

[0503] The IL has three call instructions that are used to transfer new argument values to a destination method. Under normal circumstances, the called method will terminate and return control to the calling method.

[0504] call is designed to be used when the destination address is fixed at the time the IL is linked. In this case, a method reference is placed directly in the instruction. This is comparable to a direct call to a static function in C. It can be used to call static or instance methods or the (statically known) superclass method within an instance method body.

[0505] calli is designed for use when the destination address is calculated at run time. A method pointer is passed on the stack and the instruction contains only the call site description.

[0506] callvirt, part of the IL VOS instruction set, uses the class of an object (known only at runtime) to determine the method to be called. The instruction includes a method reference, but the particular method isn't computed until the call actually occurs. This allows an instance of a subclass to be supplied and the method appropriate for that subclass to be invoked. The callvirt instruction is used both for instance methods and methods on interfaces.

[0507] In addition, each of these instructions can be immediately preceded by a tailcall instruction. This specifies that the calling method terminates with this method call (and returns whatever value is returned by the called method). The tailcall instructs the JIT compiler to discard the calling stack frame prior to making the call (if the call is from untrusted code to trusted code the frame cannot be fully discarded for security reasons). When the called method executes a ret instruction, control returns not to the calling method but rather to wherever that method would itself have returned. Notice that the tailcall instruction shortens the lifetime of the caller's frame so it is incorrect to pass managed pointers (type &) as arguments.

[0508] Finally, there are two instructions that indicate an optimization of the tailcall case:

[0509] jmp is followed by a methodref token and indicates that the current method's stack frame should be discarded, its arguments should be transferred intact to the destination method, and control should be trans-

ferred to the destination. The signature of the calling method must exactly match the signature of the destination method.

[0510] `jmp` takes a computed destination address on the stack, pops it off the stack, discards the current stack frame, transfers the current arguments to the destination method, and transfers control to the destination method. The signature of the calling method must exactly match the signature of the destination method.

[0511] Method Calls: Computed Destinations

[0512] The destination of a method call can be either encoded directly in the IL instruction stream (the `call` and `jmp` instructions) or computed (the `callvirt`, `calli`, and `jmp` instructions). The destination address for a `callvirt` instruction is automatically computed by the Execution Engine based on the method token and the value of the first argument (the `this` pointer). The method token must refer to a virtual method on a class that is a direct ancestor of the class of the first argument. The EE computes the correct destination by, effectively, locating the nearest ancestor of the first argument's class that supplies an implementation of the desired method (the implementation can be assumed to be more efficient than the linear search implied here).

[0513] For the `calli` and `jmp` instructions the IL code is responsible for computing a destination address and pushing it on the stack. This is typically done through the use of a `ldftn` or `ldvirtfn` instruction at some earlier time. The `ldftn` instruction includes a metadata token in the IL stream that specifies a method, and the instruction pushes the address of that method. The `ldvirtfn` instruction takes a metadata token for a virtual method in the IL stream and an object on the stack. It performs the same computation described above for the `callvirt` instruction but pushes the resulting destination on the stack rather than calling the method.

[0514] The `calli` instruction includes a call site description that includes information about the native calling convention that should be used to invoke the method. The EE does not check that this correctly matches the calling convention for the method that is being called; any mismatch will result in unpredictable behavior. The `jmp` instruction requires that the destination method have the same calling convention and the method that contains the `jmp` instruction; any mismatch will result in unpredictable behavior.

[0515] Method Calls: Virtual Calling Convention

[0516] The IL provides a "virtual calling convention" that is converted by the JIT into a native calling convention. The JIT determines the optimal native calling convention for the target architecture. This allows the native calling convention to differ from machine to machine, including details of register usage, local variable homes, copying conventions for large call-by-value objects (as well as deciding, based on the target machine, what is considered "large"). This also allows the JIT to reorder the values placed on the IL virtual stack to match the location and order of arguments passed in the native calling convention.

[0517] The EE uses a single uniform calling convention for all method calls. It is the responsibility of the JITters to convert this into the appropriate native calling convention.

[0518] The virtual calling convention is:

[0519] If the method being called is an instance method (class or interface) or a virtual method, first push the `this` pointer. For methods on Objects (including boxed value classes), the `this` pointer is of type `O` (object reference). For methods on value classes, the `this` pointer is provided as a by-ref parameter; that is, the value is a pointer (managed, `&`, or unmanaged, `*` or `I`) to the instance.

[0520] If the return value from the method is a value class, a home (a managed or unmanaged pointer) for the value is pushed next. The home must be supplied even if the caller will discard the value. The method's signature will have a specially marked parameter that corresponds to this argument.

[0521] Push the remaining arguments in left-to-right order (that is, push the leftmost argument first). Execute the appropriate call instruction (`call`, `calli`, or `callvirt` any of which may be preceded by `tailcall`).

[0522] Method Calls: Parameter Passing

[0523] The EE supports three kinds of parameter passing, all indicated in metadata as part of the signature of the method. Each parameter to a method has its own passing convention (e.g., the first parameter may be passed by-value while all others are passed by-ref).

[0524] Parameter may be passed as follows:

[0525] By-value parameters, where the value of an object is passed from the caller to the callee.

[0526] By-ref parameters, where the address of the data is passed from the caller to the callee, and the type of the parameter is therefore a managed or unmanaged pointer.

[0527] Ref-any parameters, where a runtime representation of the data type is passed along with the address of the data, and the type of the parameter is therefore one specially supplied for this purpose.

[0528] The verifier checks that the types of parameters match the types of values passed, but is otherwise unaware of the details of the calling convention.

[0529] For primitive types (integers, floats, etc.) the caller copies the value onto the stack before the call. For Objects the object reference (type `O`) is pushed on the stack. For managed pointers (type `&`) or unmanaged pointers (type `U`), the address is passed from the caller to the callee. This allows the JITter to determine when the copy is made: by the caller, by the callee as part of the method prolog, or immediately prior to any mutation operations.

[0530] By-Ref Parameters are the equivalent of C++ reference parameters or PASCAL var parameters: instead of passing as an argument the value of a variable, field, or array element, its address is passed instead; and any assignment to the corresponding parameter actually modifies the corresponding variable, field, or array element. Much of this work is done by the higher-level language, which hides from the user the need to compute addresses to pass a value and the use indirection to reference or update values.

[0531] Passing a value by reference requires that the value have a home and it is the address of this home that is passed. Constants and intermediate values cannot be passed as by-ref parameters because they have no home.

[0532] Some addresses (e.g., local variables and arguments) have lifetimes tied to the method contexts. These cannot be correctly referenced outside their lifetimes, and so they should not be stored in locations that last beyond their own lifetime. The IL does not (and cannot) enforce this restriction, so the IL generator must enforce this restriction or the resulting IL will not work correctly. For code to be verifiable by-ref parameters may only be passed to other methods or referenced via the appropriate stind or ldind instructions.

[0533] By-ref parameters and value classes are sufficient to support statically typed languages (Java, C++, Pascal, etc.). They also support dynamically typed languages that pay a performance penalty to box value classes before passing them to polymorphic methods (Lisp, Scheme, SmallTalk, etc.). Unfortunately, they are not sufficient to support languages like Visual Basic that require by-reference passing of unboxed data to methods that are not statically restricted as to the type of data they accept. These languages require a way of passing both the address of the home of the data and the static type of the home. This is exactly the information that would be provided if the data were boxed, but without the heap allocation required of a box operation.

[0534] Ref-Any parameters address this requirement. A ref-any parameter is very similar to a standard by-ref parameter but the static data type is passed as well as the address of the data. Like by-ref parameters, the argument corresponding to a ref-any parameter must have a home. If it were not for the fact that the verifier and the memory manager must be aware of the data type and the corresponding address, a by-ref parameter could be implemented as a standard value class with two fields: the address of the data and the type of the data. Like a regular by-ref parameter, a ref-any parameter can refer to a home that is on the stack, and that home will have a lifetime limited by the call stack. Thus, the IL generator must appropriate-checks on the lifetime of by-ref parameters; and the verifier imposes the same restrictions on the use of ref-any parameters as it does on by-ref parameters.

[0535] A ref-any parameter is passed either by creating a new ref-any using the mkrefany instruction or copying an existing one using ldrefany. Given a ref-any argument, the address to which it can be referred can be loaded using the ldanyaddr instruction (which also tests that the type matches the desired type), and the value can then be extracted or modified using the ldind.* or stind.* instructions. The type associated with the ref-any parameter can be extracted using the ldanytype instruction.

[0536] A given parameter can be passed using any one of the parameter passing conventions: by-value, by-ref, or ref-any. No combination of these is allowed for a single parameter, although a method may have different parameters with different calling mechanisms.

[0537] There are a pair of non-obvious facts about the parameter passing convention:

[0538] A parameter that has been passed in as ref-any cannot be passed on as by-ref or by-value without a runtime type check and (in the case of by-value) a copy.

[0539] A by-ref parameter can be passed on as a ref-any by attaching the static type.

[0540] The table of FIG. 27 illustrates the parameter passing convention used for each data type.

[0541] Exception Handling

[0542] The EE supports an exception handling model based on the concept of exceptions object and "protected blocks" of code. When an exception occurs, an object is created to represent the exception. All exceptions objects are boxed instances of some subclass of Microsoft.Runtime.Exception. Users can create their own exception classes by subclassing Microsoft.Runtime.Exception.

[0543] There are three kinds of protected blocks:

[0544] A finally clause which must be executed whenever the block exits, regardless of whether that occurs by normal control flow or by an unhandled exception.

[0545] A type-filtered handler which handles any exception that is of a specified class or any of its sub-classes.

[0546] A user filtered handler which runs a user-specified set of IL instructions to determine whether the exception should be ignored (i.e. execution should resume), handled by the associated handler, or passed on to the next protected block.

[0547] EE instructions can throw the following exceptions as part of executing individual instructions. The documentation on a particular instruction will list all the exceptions the instruction can throw (except for the general purpose ExecutionEngineException described below that can be generated by all instructions). The list of exceptions in one embodiment is:

[0548] Base Instructions

[0549] ArithmeticException

[0550] DivideByZeroException

[0551] ExecutionEngineException

[0552] InvalidAddressException

[0553] SecurityException

[0554] StackOverflowException

[0555] Object Model Instructions

[0556] ClassloadException

[0557] IndexOutOfRangeException

[0558] InvalidAddressException

[0559] InvalidCastException

[0560] MissingFieldException

[0561] MissingMethodException

[0562] NullReferenceException

[0563] OutOfMemoryException

[0564] SecurityException

[0565] StackOverflowException

[0566] SynchronizationLockException

[0567] The `ExecutionEngineException` is special. It can be thrown by any instruction and indicates an unexpected inconsistency in the EE. In one embodiment, code that has been passed through the code verifier should never throw this exception (it is a defect in either the verifier or the EE if it does), unless it is thrown explicitly by the code sequence. However, unverified code can cause this error if the code is corrupt or inconsistent in some way.

[0568] Note that, because of the verifier, there are no exceptions for things like 'MetadataTokenNotFound.' The verifier can detect this inconsistency before the instruction is ever executed (the code is then considered unverified). If the code has not been verified, this type of inconsistency would raise the generic `ExecutionEngineException`.

[0569] Exceptions can also be thrown by the COM+ Runtime, as well as by user code, using the throw instruction. The handing of an exception is identical, regardless of the source.

[0570] Each method in an executable has associated with it a (possibly empty) array of exception handling information. Each entry in the array describes a protected block, its filter, and its handler (which may be either a catch handler or a finally handler). When an exception occurs, the EE searches the array for the first protected block that

[0571] Protects a region including the current instruction pointer and

[0572] Is a catch handler block and

[0573] Whose filter wishes to handle the exception

[0574] If a match is not found in the current method, the calling method is searched, and so on. If no match is found the EE will dump a stack trace and abort the program. If a match is found, the EE walks the stack back to the point just located, but this time calling the finally handlers. It then starts the corresponding exception handler. Stack frames are discarded either as this second walk occurs or after the handler completes, depending on information in the exception handler array entry associated with the handling block.

[0575] Some things to notice are:

[0576] The ordering of the exception clauses is important. If handlers are nested, the most deeply nested try blocks must come before the try blocks that enclose them.

[0577] Exception handlers can access the local variables and the local memory pool of the routine that catches the exception, but any intermediate results on the operand stack at the time the exception was thrown are lost.

[0578] Execution cannot be resumed at the location of the exception. This restriction may be relaxed in the future.

[0579] Furthermore, the IL has special instructions to:

[0580] Throw a user-defined exception.

[0581] Leave a protected block and execute the appropriate finally clauses within a method, without throwing an exception.

[0582] End a user-supplied filter clause and return a value indicating whether to handle the exception

[0583] End a catch handler and return to normal processing (possibly unwinding the stack)

[0584] End a finally clause and continue crawling the stack

[0585] It is not legal to enter a user-supplied filter, catch handler, or finally handler other than through the exception mechanism. It is also not legal to exit a protected region with any control transfer instruction other than a leave.

[0586] OptIL: An Instruction Set Within IL

[0587] A fundamental issue associated with generating intermediate IL is how much of the work is done by the IL generator and how much of the work is done by the Execution Engine (via a JIT compiler). The IL instruction set was designed to be easy for compilers to generate so that IL can be generated quickly in rapid application development (RAD) environments, where compile speed and ease of debugging are at a premium.

[0588] On the other hand, in situations where load time is important, it is useful to do as much work as possible in the code generator, before the executable is loaded. In particular it is useful to do expensive optimizations like common sub-expression elimination, constant folding, loop restructuring, and even register allocation in the code generator (as would be done in a traditional compiler). The instruction set should be able to represent such optimized code as well.

[0589] Finally, in some environments it is important that the JITter be small and run in a nearly constant amount of memory, even for large methods. The instruction set should allow a compiler to compute information and pass it on to the JITter that will reduce the memory required by the JITter (e.g., register allocation and branch targets).

[0590] In the COM+ Runtime environment, an optimizing compiler can best express many optimizations by generating OptIL. OptIL is optimized code represented using the same IL instruction set; however, OptIL differs from non-OptIL code in the following ways

[0591] Many transformations will have been done (e.g., loop restructuring, constant folding, CSE).

[0592] The code will obey certain conventions (e.g., method calls are not nested).

[0593] There will be additional annotations (e.g., exactly when each variable is used for the last time).

[0594] Note that an OptIL program is still a valid IL program (it can be run by the normal EE), but because it has been optimized by the code generator it can be compiled to native code very quickly and using little memory.

[0595] Conclusion

[0596] Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that any arrangement which is calculated to achieve the same purpose may be substituted for the specific embodiments shown. This application is intended to cover any adaptations or variations of the present invention. Therefore, it is manifestly intended that this invention be limited only by the following claims and equivalents thereof.

We claim:

1. A computer-implemented method comprising:
 - verifying metadata of an intermediate language code for consistency and accuracy; and,
 - verifying the intermediate language code for consistency and accuracy.
2. The method of claim 1, further comprising executing the intermediate language code upon verification of the code and the metadata thereof, by translating the intermediate language code into executable code.
3. The method of claim 1, further comprising initially generating the intermediate language code by compiling a source code into the intermediate language code.
4. The method of claim 1, wherein verifying the intermediate language code for consistency and accuracy comprises:
 - performing a syntactic check of the intermediate language code; and,
 - performing a semantic check of the intermediate language code.
5. The method of claim 4, wherein performing a semantic check comprises checking reference aspects of the intermediate language code.
6. The method of claim 5, wherein the reference aspects comprise at least one of byrefs and refany's.
7. The method of claim 4, wherein performing a semantic check comprises checking value classes of the intermediate code language.
8. The method of claim 4, wherein performing a semantic check comprises checking native-size primitive types of the intermediate code language.
9. The method of claim 4, wherein performing a semantic check comprises verifying tail calls of the intermediate code language.
10. A machine-readable medium having processor instructions stored thereon for execution by a processor to perform a method comprising:
 - verifying metadata of an intermediate language code for consistency and accuracy; and,
 - verifying the intermediate language code for consistency and accuracy.
11. The medium of claim 10, further comprising executing the intermediate language code upon verification of the code and the metadata thereof, by translating the intermediate language code into executable code.
12. The medium of claim 10, further comprising initially generating the intermediate language code by compiling a source code into the intermediate language code.
13. The medium of claim 10, wherein verifying the intermediate language code for consistency and accuracy comprises:
 - performing a syntactic check of the intermediate language code; and,
 - performing a semantic check of the intermediate language code.
14. The medium of claim 13, wherein performing a semantic check comprises checking reference aspects of the intermediate language code.
15. The medium of claim 14, wherein the reference aspects comprise at least one of byrefs and refany's.
16. The medium of claim 13, wherein performing a semantic check comprises checking value classes of the intermediate code language.
17. The medium of claim 13, wherein performing a semantic check comprises checking native-size primitive types of the intermediate code language.
18. The medium of claim 13, wherein performing a semantic check comprises verifying tail calls of the intermediate code language.
19. A computerized system comprising:
 - an execution engine to translate an intermediate language code to an executable code based thereon for execution; and,
 - a verifier, part of the execution engine, to verify the intermediate language code and metadata of the intermediate language code prior to translation thereof to executable code.
20. The system of claim 19, wherein the verifier verifies the intermediate language code by performing a syntactic check and a semantic check thereof.
21. The system of claim 19, wherein at least one of the execution engine and the verifier comprises a computer program executed by a processor from a computer-readable medium.
22. A computerized device comprising:
 - a processor;
 - a computer-readable medium;
 - an intermediate code stored as data on the medium;
 - an executable code based on the intermediate code and stored as data on the medium;
 - an execution engine executed by the processor from the medium to translate the intermediate code to the executable code for execution; and,
 - a verifier, part of the execution engine, to verify the intermediate language code and metadata of the intermediate language code prior to translation thereof to executable code.
23. A computerized system comprising:
 - a first component of an intermediate language code verifier to perform semantic checks on an intermediate-language code, including verifying that branch destinations within the code are proper; and,
 - a second component of the intermediate language code verifier to perform semantic checks on the intermediate-language code.
24. The system of claim 23, where at least one of the first component and the second component is executed by a processor from a computer-readable medium.
25. The system of claim 23, wherein the first component further is to determine a location of each of a plurality of basic blocks within the intermediate-language code.
26. The system of claim 25, wherein the first component further is to prepare a sorted list of the location of each of the plurality of basic blocks.

27. The system of claim 23, wherein for each of a plurality of opcodes within the intermediate language code, the second component further is to perform a corresponding list of operations and checks.

28. The system of claim 23, wherein the intermediate language code is governed by an assumption that a type of an argument to a method of the code is fixed.

29. The system of claim 23, wherein the intermediate language code is governed by an assumption that a type of a primitive local variable of the code is fixed.

30. The system of claim 29, wherein the type of the primitive local variables comprises one of an integer and a floating point.

* * * * *